



UNIVERSITÀ
DEGLI STUDI
DI PALERMO



Programmazione asincrona

Corso di Programmazione Web e Mobile
a.a. 2021/2022

Prof. Roberto Pirrone

Sommario

- Concetti di programmazione asincrona
- Web worker
- Classe Promise
 - `then()`, `catch()`, `finally()`
 - `async/await`
- Richieste asincrone al server
 - Uso di `fetch`
 - Uso di `XMLHttpRequest` (AJAX)
 - Storage locale dei dati
- Generatori

Concetti di programmazione asincrona

- La programmazione asincrona riguarda la possibilità di lanciare da programma più *thread* di lavoro paralleli e gestire i loro risultati risultati
- Questo tipo di soluzioni è molto importante in tutti i casi in cui l'onere computazionale relativo all'esecuzione della logica di controllo del client è particolarmente elevato
- Un altro ambito rilevante è quello della gestione di *richieste HTTP asincrone* verso il server

Concetti di programmazione asincrona

- In Javascript esistono diverse soluzioni per innescare una esecuzione asincrona
 - Uso di Web Worker
 - Creazione e gestione di Promise
 - Richieste asincrone al server con `fetch` o creazione di `XMLHttpRequest`

Web worker

- Un web worker è un processo separato che viene lanciato dal programma principale incorporato nella pagina HTML del client che comunica con quest'ultimo tramite *scambio di messaggi*
- La classe Worker fa parte della cosiddetta Web API di Javascript

Web worker

```
<script>  
    ... codice dello script ...  
    let worker = new Worker( 'fileJSdelWorker' );  
  
    worker.addEventListener( 'message' ,  
        function(event) {  
            ... callback che gestisce le risposte del  
            worker che si trovano in event.data ...  
        } );  
    worker.postMessage( oggettoMessaggioDati );  
</script>
```

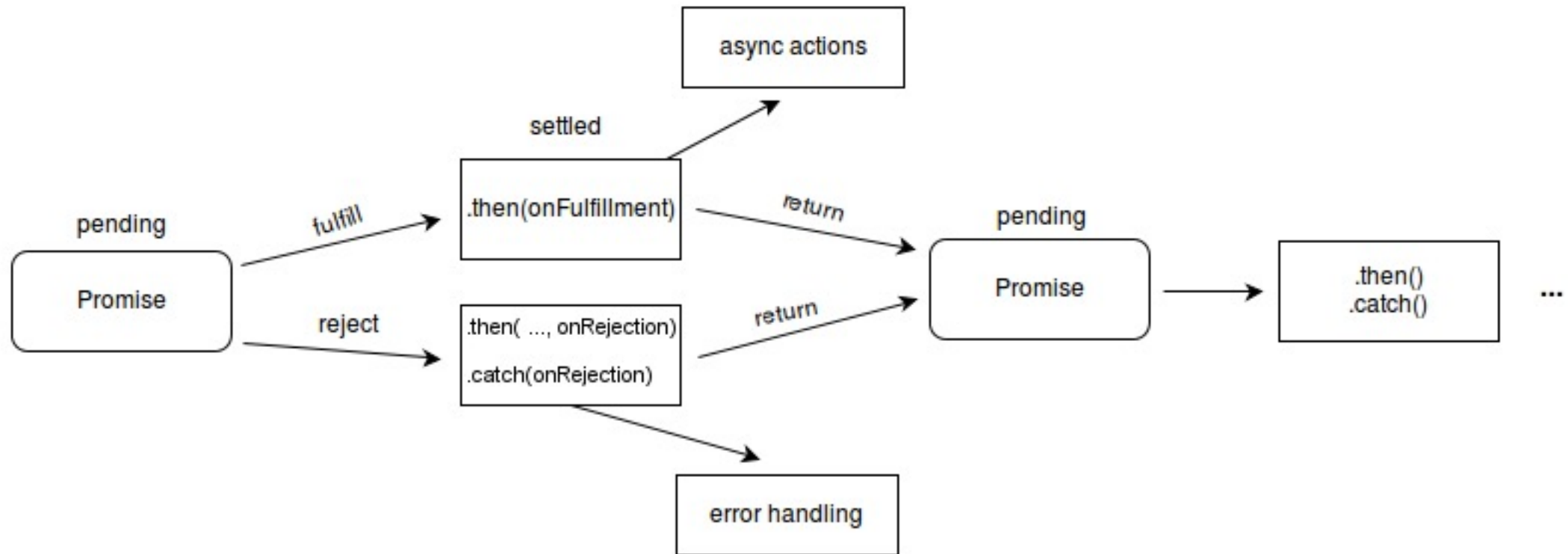
Web worker

```
// Codice del worker  
    ... codice di utilità del worker ...  
  
onmessage=function(event){  
    ... callback che gestisce le operazioni del  
    worker quando riceve un messaggio il cui  
    contenuto si trova in event.data ...  
  
    postMessage(oggettoMessaggioRisposta);  
};
```

Promise

- La classe `Promise` definisce una operazione differita che viene eseguita asincronamente rispetto al resto del programma
- La `Promise` può trovarsi in tre stati: pendente, *risolta*, cioè eseguita correttamente, ovvero *rigettata*, cioè fallita nella sua esecuzione
- `Promise` definisce anche dei metodi per gestire l'esecuzione dopo la risoluzione/rigetto ovvero per gestire l'insorgere di un errore

Promise



Promise

- Le callback `resolve(value)` e `reject(reason)` consentono di risolvere/rigettare esplicitamente una `Promise`
- Gli argomenti di `resolve` e `reject` sono passati all'oggetto `Promise` mentre viene restituito un oggetto `Promise` risolto/rigettato
- I metodi `then()`, `catch()` e `finally()` gestiscono una `Promise` dopo che è stata risolta/rigettata e possono essere invocati usando l'operatore `'.'` su un oggetto `Promise` per cui è stata eseguita `resolve` o `reject`

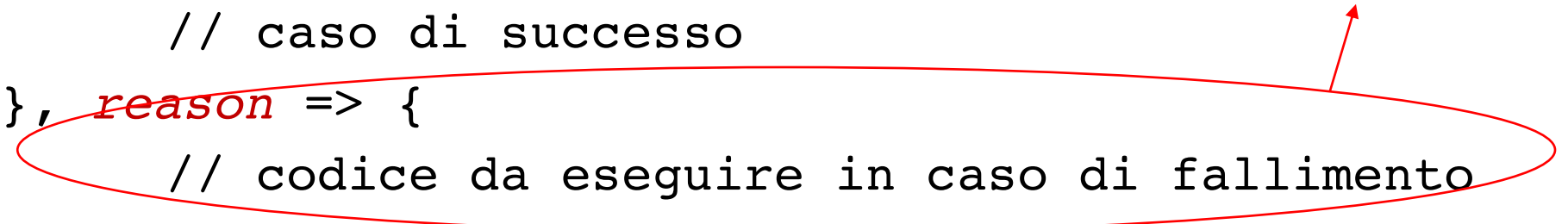
Promise

```
const myprom = new Promise(  
  (resolveFun, rejectFun) => {  
    // codice asincrono che include la  
    // chiamata a resolveFun(value) in  
    // caso di successo e rejectFun(reason)  
    // in caso di fallimento  
  });
```

Promise

```
myprom.then(value => {  
    // codice da eseguire in  
    // caso di successo  
}, reason => {  
    // codice da eseguire in caso di fallimento  
}).catch(errore => {  
    // codice gestione di errore  
}).finally(() => {  
    // codice da eseguire  
    // comunque alla fine  
});
```

Argomento opzionale



Promise

- La chiamata dei metodi `then()`, `catch()` e `finally()` usando l'operatore `'.'` è certamente poco leggibile
- Si può rendere esplicito il fatto che un oggetto verrà valorizzato con il risultato di una `Promise` risolta e/o rigettata usando la parola chiave `await` nell'assegnamento
- L'assegnamento con `await` può essere utilizzato *solamente* nello scope di una funzione dichiarata esplicitamente asincrona con la direttiva `async function`

Promise

```
async function myAsyncFunction (myArgumentList) {  
  
    let myPromiseResolved =  
        await functionReturningAPromise(...);  
  
    // codice che usa l'oggetto  
    // myPromiseResolved  
  
}
```

Richieste asincrone al server

- Una parte fondamentale dell'operato del client in una web application è quello di gestire delle richieste al server per ottenere ulteriori risorse informative e/o inviare dati
- Le richieste al server sono per definizione delle chiamate asincrone perché dipendono dal traffico sulla rete e dalle condizioni del server
- Le modalità principali per richiedere risorse al server sono
 - `fetch()` che è parte della Javascript Web API
 - Uso dell'oggetto `XMLHttpRequest` (tecnologia AJAX – *Asynchronous Javascript And XML*)

Richieste asincrone al server

```
fetch('percorso/della/risorsa', {  
    // oggetto opzionale di classe Headers  
    // con i dati della richiesta  
}).then(response => {  
    // codice di gestione della risposta contenuta  
    // nell'oggetto response  
    return response.json(); // questa è una Promise  
    // di un oggetto json  
  
    // si potrebbe usare response.text(),  
    // response.blob(), response.arrayBuffer() e  
    // response.formData() per i rispettivi formati  
  
    // response.headers è l'oggetto di classe Headers  
    // con gli header della risposta  
}).then(datiRisposta => {  
    // codice di gestione dei  
    // dati della risposta}).catch(error => {  
        console.log(error.message);  
    });
```


Richieste asincrone al server

```
// formato dell'oggetto di richiesta
{
    method:      // 'GET', 'HEAD', 'POST' ...
    headers:     // oggetto Headers contenente coppie chiave
                  // valore con gli header HTTP impostabili
                  // con i metodi get() e set()
    body:        // oggetto contenente coppie chiave-valore
                  // con i dati da inviare al server
    credentials: // 'omit', 'same-origin',
                  // 'include'

    ...
}
```

Richieste asincrone al server

```
var xhttp = new XMLHttpRequest();

// eventuali impostazioni della richiesta con setRequestHeader()

xhttp.onreadystatechange = function() {
    try {
        if (this.readyState == 4 && this.status == 200) {
            // codice di gestione della disposta
            // this.responseText e this.responseXML
            // contengono la risposta se testo o documento
            // this.response è l'oggetto risposta se
            // si tratta di dati binari
        }
    } catch (error) {
        // codice di gestione dell'errore
    }
};

xhttp.open('GET', 'percorso/della/risorsa', true/false);
xhttp.send([oggetto corpo della richiesta]);
```

Gli stati vanno da 0 a 4
4 → documento caricato

getResponseHeader()
consente di accedere agli
header della risposta

Richiesta
asincrona/sincrona

Richieste asincrone al server

- Le risposte del server contengono spesso dati che può essere comodo conservare localmente sul client:
 - Dati di elevate dimensioni che servono come database locale per il client
 - Dati di sessione dell'utente
- In Javascript ci sono due oggetti analoghi per questo fine
 - `localStorage`
 - `sessionStorage`, dedicato proprio ai dati di sessione, indicativamente fino allo spegnimento del browser

Richieste asincrone al server

// vale anche per sessionStorage

```
localStorage.setItem( 'chiave' , 'stringa valori' )
```

```
localStorage.getItem( 'chiave' )
```

Generatori

- I generatori sono funzioni che alla fine dell'esecuzione restituiscono un risultato, ma restano sospese per riprendere l'elaborazione a partire dall'ultimo risultato prodotto
- Un generatore è dichiarato con `function*` e restituisce i propri risultati con `yield`
- `yield*` si usa per restituire un risultato iterabile, elemento per elemento, ovvero per delegare l'esecuzione a un altro generatore

Generatori

- Il generatore va assegnato ad un oggetto iteratore che, invocando il proprio metodo `next ()`, innescherà una esecuzione del generatore e ne aggiornerà lo stato interno usando come parametro il valore restituito dall'ultima chiamata `yield`
- Una invocazione di `next (valore)` usa *valore* per innescare l'esecuzione, al posto di ciò che aveva restituito `yield`

Generatori

```
function* myGenerator(parametri) {  
    // codice del generatore  
    yield risultato;  
}
```

```
let gen = myGenerator(parametriDiInvocazione);  
gen.next()    // prima esecuzione  
gen.next()    // seconda esecuzione a partire  
              // dal primo valore di risultato
```

...