UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze e Metodi
dell'Ingegneria

Tecnologie Web e Internet of Things

Stefania Monica
stefania.monica@unimore.it

JavaScript Asincrono

Tecnologie Web e Internet of Things

- JavaScript nasce per realizzare applicazioni interattive
- Un'applicazione interattiva ha le seguenti caratteristiche
  - L'applicazione è per la maggior parte del tempo in attesa di eventi (da parte degli utenti o della rete)
  - Gli eventi accadono indipendentemente dallo stato dell'applicazione e quindi vengono detti asincroni
- Si noti che non tutte le applicazioni possono essere considerate interattive
  - Ad esempio, un'applicazione di calcolo scientifico, che tiene costantemente impegnata la CPU per svolgere calcoli

- Quindi, recentemente, l'organismo ECMA (*European Computers Manufacturers Association*) promuove degli standard per JavaScript, cercando di migliorare le sue caratteristiche nella realizzazione di applicazioni interattive.

Tecnologie Web e Internet of Things

- Facciamo riferimento a due versioni di JavaScript che comunemente vengono dette ECMAScript 5 e 6
  - Miglioramenti dal punto di vista del linguaggio di programmazione
  - Miglioramenti alle strutture dati di uso comune
  - Supporto al cosiddetto JavaScript asincrono
- Per JavaScript asincrono si intende tutto il supporto offerto dal linguaggio per la gestione efficace di eventi asincroni
  - Il supporto offerto da JavaScript asincrono per le comunicazioni verso i server viene spesso chiamato AJAX (*Asynchronous JavaScript And XML*)
  - AJAX viene introdotto prima delle estensioni del linguaggio e fa riferimento a XML, quindi oggi non viene più utilizzato nella sua versione originale

Miglioramenti del Linguaggio di Tipo General Purpose

- `"use strict";`
  - Scrivere `"use strict";` all'inizio di un blocco o di uno script forza il compilatore JavaScript ad essere rigido
  - Ad esempio, le variabili vanno sempre dichiarate
  - Si tratta di un aiuto per aumentare la qualità del codice
- `let` e `const`
  - Permettono di definire variabili e costanti locali a script o a blocchi
  - Consentono di riusare i nomi delle variabili legandole ai blocchi
- Moduli
  - Permettono di raggruppare funzioni e altri strumenti in un unico spazio di nomi
  - Permettono di decidere in modo fine cosa importare e cosa esportare
  - Permettono di inizializzare le funzionalità del modulo
- JavaScript asincrono
  - I micro-task e le promise

Ciclo di Esecuzione e Coda degli Eventi

- JavaScript è pensato per le applicazioni interattive, ma:
  - Gli scenari di riferimento di JavaScript sono applicazioni piccole e semplici
  - Tradizionalmente si vuole permettere anche a programmatori non esperti di utilizzare JavaScript per arricchire l'esperienza del Web
- Quindi questi vincoli impongono alcune scelte di base fatte dai progettisti di JavaScript
  - Si vuole impedire la programmazione concorrente a causa di tutti i problemi legati alla condivisione dei dati
  - Si vuole lasciare al browser la possibilità di ottimizzare l'esecuzione del codice JavaScript
- Quindi, il ciclo di esecuzione di un'applicazione JavaScript, che utilizza sempre la coda degli eventi, viene motivato dalle considerazioni precedenti

Ciclo di Esecuzione e Coda degli Eventi

- Un'applicazione JavaScript per il Web viene eseguita in un unico processo:
  - Il processo è collegato al tab che ha caricato la pagina Web da cui è stato caricato JavaScript
  - Il processo non è multi-thread
- Il processo di esecuzione dispone di una coda detta coda degli eventi (più propriamente, coda dei micro-task)
- Il ciclo di esecuzione di un'applicazione JavaScript è il seguente:
  - Se la coda degli eventi è vuota, mettiti in attesa che arrivi un micro-task nella coda
  - Scoda il primo micro-task dalla coda degli eventi
  - Esegui il micro-task appena scodato
  - Aggiorna la visualizzazione della pagina Web
  - Torna all'inizio

Ciclo di Esecuzione e Coda degli Eventi

- Quindi:
  - I micro-task, che sono realizzati in JavaScript, devono essere veloci
  - Quando l'utente interagisce con un elemento grafico, il browser genera un micro-task che viene accodato
  - Il micro-task relativo a un evento si limita a eseguire il codice dell'handler dell'evento
  - Se i micro-task richiedono tanto tempo, il browser si accorge che i micro-task collegati agli eventi rimangono in coda troppo tempo e quindi, dopo averlo richiesto all'utente, chiude il processo collegato alla coda degli eventi
- Tutte queste considerazioni, se non si sfruttano le funzionalità di JavaScript asincrono, richiedono di parcellizzare il codice in tanti micro-task di durate brevi
  - Complicando il testing
  - Potenzialmente introducendo anomalie

Example0

- Questo esempio ha lo scopo di fare un conto alla rovescia da 10 a 0 e visualizzare i valori nella pagina HTML
  - La pagina contiene uno script che definisce una funzione go
  - La pagina contiene un handler dell'evento onload
  - Quando il browser ha completato il caricamento della pagina genera un micro-task corrispondente all'esecuzione dell'handler dell'evento onload e lo mette nella coda degli eventi
  - Quando questo micro-task viene rimosso dalla coda degli eventi per essere eseguito invoca la funzione go
  - La funzione go conta da 10 a 0 e modifica il contenuto della pagina

- I cambiamenti nella pagina non si vedono perché la pagina viene aggiornata solo quando il micro-task che esegue go viene completato

Tecnologie Web e Internet of Things

- Per risolvere il problema precedente bisogna spezzare il codice in micro-task in modo che ogni variazione del conto alla rovescia sia in un micro-task separato
  - Facendo così, ogni variazione del conto alla rovescia viene visualizzata
- Un modo per mettere un micro-task nella coda degli eventi è utilizzare `setTimeout`, che vuole due parametri:
  - La funzione da eseguire quando il micro-task viene tolto dalla coda
  - L'attesa in millisecondi prima di inserire il micro-task nella coda degli eventi
- Nota: L'attesa può anche essere nulla e quindi `setTimeout` può anche essere usato per manipolare la coda degli eventi senza forzatamente rallentare l'esecuzione

Funzioni Anonime

- Funzione anonima o lambda function o lambda expression
- Una funzione anonima è una funzione senza nome e può essere definita nel modo seguente:

```
(function () {
    //…
});
```

- Esempio con assegnamento:

```
let show = function() {
    console.log('Anonymous function');
};

show();
```

Funzioni Anonime

- Le funzioni anonime sono spesso passate come argomenti ad altre funzioni
- Esempio:

```
setTimeout(function() {
    console.log('Execute later after 1 second')
}, 1000);
```

- La funzione anonima è passata alla funzione setTimeOut, che la esegue un secondo dopo
- Le funzioni anonime possono essere scritte anche come arrow functions

Funzioni Anonime

- L'esempio precedente:

```
let show = function () {
    console.log('Anonymous function');
};
```

può essere riscritto come:

```
let show = () => console.log('Anonymous function');
```

- Altro esempio:

```
let add = function (a, b) {
    return a + b;
};

let add = (a, b) => a + b;
```

Example1

- La funzione che viene associata a un micro-task messo in coda da uno script (ad esempio, con `setTimeout`) viene chiamata (funzione) callback
  - Lo script dà una funzione e il browser “chiama indietro” lo script quando esegue questa funzione
  - Le callback sono funzioni qualsiasi e quindi esistevano già prima di JavaScript asincrono

Promise

- ECMAScript 6 introduce il meccanismo delle promise per offrire le funzionalità asincrone di JavaScript.
- Ogni promise può essere in uno dei tre stati seguenti:
  - In attesa (di essere soddisfatta)
  - Soddisfatta (risolta)
  - Fallita (rigettata)
- Quando una promise viene costruita:
  - Riceve come argomento la funzione da utilizzare per creare il micro-task
  - Questa funzione ha due argomenti, che sono la funzione da chiamare per entrare nello stato Soddisfatta (resolver) e quella per entrare nello stato Fallita (rejecter)
  - Entra nello stato In attesa e chiama la funzione precedente
  - Quando il micro-task mette la promise in stato Soddisfatta, viene chiamata la funzione passata come argomento a `then`
  - Quando il micro-task mette la promise in stato Fallita, viene chiamata la funzione passata come argomento a `catch`

Example2

- La funzione `sleep` costruisce una promise e la ritorna
- L'argomento della costruzione della promise è una funzione con due argomenti: `resolver` e `rejecter`
- L'argomento della costruzione crea un micro-task con `setTimeout` e, quando viene eseguito (dopo il timeout) mette la promise in stato Soddisfatta
- Sulla promise ritornata da `sleep` viene usato `then` per specificare la funzione da chiamare quando la promise va in stato Soddisfatta
- Dopo `then`, si potrebbe usare `catch` per specificare la funzione da chiamare quando la promise va in stato Fallita
  - Ad esempio, se la promise non è un timeout ma è un tentativo di scaricare dalla rete

Async e Await

- Le promise non richiedono dei cambiamenti a JavaScript perché sfruttano le callback
  - L'argomento di `then` è una callback da usare quando la promise va in stato Soddisfatta
  - L'argomento di `catch` è una callback da usare quando la promise va in stato Fallita
- ECMAScript 6 introduce dei cambiamenti al linguaggio per gestire in modo più efficace le promise
- Parola chiave `await`:
  - Riceve come argomento una promise
  - Blocca l'esecuzione finché la promise non va in stato Soddisfatta o Fallita
- Parola chiave `async`:
  - Solo le funzioni dichiarate async (asincrone) possono utilizzare `await`

Async e Await

- Si noti che una funzione asincrona viene sempre chiamata all'interno di una sequenza di chiamate che derivano da un micro-task associato a un evento
- Quindi, non è possibile che la await si blocchi davvero, altrimenti bloccherebbe la coda degli eventi
- Infatti, quello che davvero fa la await è
  - Costruire un nuovo micro-task che contenga tutto ciò che segue la await stessa nella funzione asicrona
  - Associare questo micro-task al cambiamento di stato della promise in Soddisfatta
- Per chiarire, si può dire che tutto quello che segue la await viene messo in una funzione che viene passata come argomento della then sulla promise su cui è stata chiamata la await

Example3

Promise e Loro Valori

- Una promise può essere associata a:
  - Un valore che viene prodotto quando la promise viene soddisfatta, oppure
  - Un'eccezione che viene prodotta quando la promise fallisce
- Questi valori possono essere passati, rispettivamente, alla funzione `resolver` e alla funzione `rejecter`
- Se la promise viene gestita con `then` e `catch`, i due valori precedenti vengono passati, rispettivamente, alle funzioni usate come argomento di `then` e `catch`
- Se la promise viene gestita con `await`:
  - Il valore associato al soddisfacimento viene ritornato dalla `await`
  - L'eccezione associata al fallimento viene lanciata dalla `await`

Example4

Example5

- Si noti che le promise possono essere usate per gestire qualsiasi evento
  - Non solo i timeout impostati con `setTimeout`
  - Ma anche gli eventi generati dall'utente
- Per gestire gli eventi dell'utente dobbiamo:
  - Associare un handler per gestire l'evento
  - Mettere la promise in stato Soddisfatta quando l'utente genera l'evento
  - Si noti che l'handler dell'evento è utile solo nell'ambito di una promise
- Per associare e de-associare handler a eventi è possibile utilizzare:
  - `addEventListener`
  - `removeEventListener`
  - Il primo argomento è il nome dell'evento (e.g., click)
  - Il secondo argomento è l'handler da utilizzare per gestire l'evento

Example5

- La funzione `nextClick` ritorna una promise che viene soddisfatta quando l'utente preme sul bottone con id `next`
  - Quando la promise viene costruita, viene definito il micro-task da utilizzare per gestire i click sul bottone
  - Il micro-task (che è l'handler dell'evento click) rimuove se stesso e poi mette in stato Soddisfatta la promise
- La funzione `countdown` utilizza `await` su una promise ritornata da `nextClick` e quindi:
  - Si mette in attesa finché l'utente non preme il bottone
  - Continua l'esecuzione della funzione dopo che l'utente ha premuto
- Ovviamente, siccome la continuazione della funzione viene eseguita in un altro micro-task, la grafica può essere aggiornata anche se sembra che siamo bloccati in un ciclo

Fetch

Nelle versioni di JavaScript che stiamo considerando è disponibile una funzione di libreria che si chiama `fetch`

- Prende come primo argomento una URL e scarica la risorsa Web associata alla URL
- Prende come secondo argomento una mappa di valori di configurazione
- Ritorna una promise che viene messa in stato:
  - Soddisfatta, quando la risorsa Web è stata scaricata del tutto
  - Fallita, quando non è stato possibile scaricare completamente la risorsa
- Si noti che `Fetch` è molto simile a `XMLHttpRequest` però:
  - Viene definito nello standard ECMA e quindi tutti i browser moderni ce l'hanno
  - E' completamente indipendente da XML

Example6

- L'esempio permette all'utente di scrivere una URL in un campo di testo e, mediante il bottone download, viene scaricata la risorsa associata alla URL
- L'handler del click sul bottone download è una funzione asincrona che usa `await` sulla promise ritornata dalla `fetch`
- Quindi l'handler si blocca finché la promise non viene soddisfatta o fallisce
  - Se la promise viene soddisfatta, i dati sono disponibili e quindi viene visualizzata l'informazione nella pagina HTML
  - Se la promise fallisce, viene visualizzato l'errore che ha causato il fallimento nella pagina HTML

Tecnologie Web e Internet of Things

Vedremo esempi di lettura dati da Web Service

- Conversione valuta:
  https://www.exchangerate-api.com/docs/overview
  (Link per ottenere la chiave)
- Film:
  http://www.omdbapi.com
  (Link per ottenere la chiave)
- Previsioni meteo:
  https://openweathermap.org/forecast5
  (Link per ottenere la chiave)

Maggiori informazioni su ECMAScript in generale:
https://www.w3schools.com/js/js_es6.asp

In precedenza…

AJAX utilizza

- Un oggetto XMLHttpRequest, per chiedere i dati al server
- JavaScript e HTML, per visualizzare o utilizzare i dati

Sintassi per creare un oggetto JavaScript di tipo XMLHttpRequest:

```
richiesta = new XMLHttpRequest();
```

Gli oggetti di tipo XMLHttpRequest hanno i metodi

- open(metodo, url, b)
  - metodo: GET o POST
  - url
  - true (asincrona) o false (sincrona)
- send()

Tecnologie Web e Internet of Things

Gli oggetti di tipo XMLHttpRequest hanno le proprietà

- `readystate`, che si usa per ottenere lo stato della richiesta
  - 0: Non è ancora stata fatta alcuna richiesta
  - 1: E' stato eseguito .open()
  - 2: E' stato eseguito .send()
  - 3: E' in corso il download dei dati
  - 4: Fine
- `status`, che si usa per ottenere lo stato della risposta
  - 200: La richiesta ha avuto successo (elenco dei possibili stati del protocollo HTTP: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)
- `responseText`, che si usa per ottenere la risposta dal server come stringa JavaScript
- `onreadystatechange`, che si usa per lanciare una funzione quando cambia lo stato dell'oggetto

Tecnologie Web e Internet of Things

Per effettuare le richieste, si usa un protocollo standard che consente di interrogare un server mediante protocollo HTTP

```
//Crea un oggetto XMLHttpRequest
richiesta = new XMLHttpRequest();
richiesta.onreadystatechange = statoCambiato;
// Apre il collegamento specificando il tipo di richiesta
richiesta.open("GET", url, true);
// Invia la richiesta al server
richiesta.send();
```

Maggiori dettagli su XMLHttpRequest:

`https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest`