Introduzione

HTTP è un protocollo a livello di applicazione stateless (senza stato) che permette la comunicazione tra sistemi distribuiti ed è le fondamenta del moderno web. Permette la comunicazione tra una varietà di host e client e supporta un insieme di configurazioni di network. HTTP si è evoluto da un protocollo progettato per scambiare file in un ambiente di laboratorio semi-fidato in un moderno labirinto Internet che contiene immagini e video ad alta risoluzione e 3D.

La comunicazione tra host e client avviene attraverso coppie richiesta/risposta. Il client inizializza un messaggio HTTP di richiesta che viene servito tramite un messaggio di risposta HTTP in cambio.

(I VERBs). Gli Urls rivelano l'identità di un particolare host ma l'azione che dovrebbe essere eseguita sull'host è specificata tramite verbi (o metodi) HTTP.

- GET: raggiunge una risorsa esistente
- POST: crea una nuova risorsa
- PUT: aggiorna una risorsa esistente (considerata come una POST specializzata)
- DELETE: cancella una risorsa esistente (considerata come una POST specializzata)

Con URLs e verbs, i client può inizializzare una richiesta al server. In ritorno, il server risponde con codici di stato e messaggi di payloads.

- 1XX: messaggi di info
- 2XX: messaggi di successo
- 3XX: messaggi di cache
- 4XX: client error messages
- 5XX: server error messages

HTML

CSS

JS

JavaScript è un linguaggio di programmazione che permette di implementare cose complesse sulle web pages. Ogni volta che una pagina web fa molto di più che rimanere lì e visualizzare informazioni statiche da guardare, visualizzare aggiornamenti di contenuto tempestivi o mappe interattive o grafica animata 2D/3D o scorrere jukebox video, ecc., puoi scommettere che JavaScript probabilmente è coinvolto.

È il terzo strato della torta a strati delle tecnologie web standard.

È un linguaggio tipizzato dinamico (il tipo di dati diventa noto in fase di esecuzione) e un linguaggio multiparadigma (funzionale, imperativo, asincrono, ecc.).

Inoltre, JavaScript è un linguaggio non bloccante a thread singolo.

JavaScript è l'implementazione dell'ECMA Script (o ES), ovvero la specifica tecnica standard del linguaggio di scripting e mantenuta dall'ECMA International. Fondamentalmente, le implementazioni dei linguaggi (come Javascript o Actionscript) sono state progettate solo per il client e lo sviluppo web. La versione più usata è la sesta versione (o ES6).

Cosa può fare?

Memorizza valori utili all'interno di variabili Operazioni su parti di testo Esecuzione di codice in risposta a determinati eventi che si verificano su una pagina web. Gestisci gli elementi HTML Può usare le API! Esegui alcune chiamate sincronizzate o asincrone a Internet Disegna e renderizza grafica 2D o 3D Ottieni la tua posizione Riproduci video e audio

Come funziona:

- Il JavaScript viene eseguito dal motore JavaScript del browser, dopo che HTML e CSS sono stati assemblati e messi insieme in una pagina web. Ciò garantisce che la struttura e lo stile della pagina siano già presenti nel momento in cui JavaScript inizia a essere eseguito. Questa è una buona cosa, poiché un uso molto comune di JavaScript è quello di modificare dinamicamente HTML e CSS per aggiornare un'interfaccia utente.
- Se JavaScript fosse stato caricato e tentato di essere eseguito prima che HTML e CSS potessero influire, si sarebbero verificati degli errori.

Sicurezza

Ogni scheda del browser è il proprio bucket separato per l'esecuzione del codice, ciò significa che nella maggior parte dei casi il codice in ciascuna scheda viene eseguito completamente separatamente e il codice in una scheda non può influire direttamente sul codice in un'altra scheda o su un altro sito Web. È una buona misura di sicurezza, se così non fosse, i pirati potrebbero iniziare a scrivere codice per rubare informazioni da altri siti Web e altre cose così brutte.

Interpretato vs compilato

Anche i browser web lo eseguono in modalità sandbox quindi non puoi accedere direttamente al sistema host dell'utente (immagine se puoi quello che puoi installare o eliminare qualsiasi file dal computer del visitatore)

JavaScript è un linguaggio interpretato, il codice viene eseguito dall'alto verso il basso e il risultato dell'esecuzione del codice viene immediatamente restituito. Non è necessario trasformare il codice in una forma diversa prima che il browser lo esegua.

I linguaggi compilati, invece, vengono trasformati (compilati) in un'altra forma prima di essere eseguiti dal computer.

Ad esempio, C/C++ vengono compilati in un linguaggio assembly che viene quindi eseguito dal computer.

JavaScript vs Java

Javascript

- Orientato agli oggetti.

- Nessuna distinzione tra tipi di oggetti.
- L'ereditarietà avviene tramite il meccanismo del prototipo e le proprietà e i metodi possono essere aggiunti dinamicamente a qualsiasi oggetto.
- I tipi di dati variabili non vengono dichiarati (tipizzazione dinamica).
- Interpretato

Java

- Basato sulla classe.
- Gli oggetti sono divisi in classi e istanze con tutta l'ereditarietà attraverso la gerarchia delle classi.
- Le classi e le istanze non possono avere proprietà o metodi aggiunti dinamicamente.
- I tipi di dati variabili devono essere dichiarati (tipizzazione statica). Semi-compilato (o semi-interpretato)

Compe importare JavaScript

JavaScript viene applicato alla tua pagina HTML in modo simile ai CSS. Mentre CSS usa gli elementi `<link>` per applicare fogli di stile esterni e gli elementi `<style>` per applicare fogli di stile interni all'HTML, JavaScript ha bisogno solo di un amico nel mondo dell'HTML, l'elemento `<script>`.

Internal:

```
<script>
// JavaScript goes here
</script>
```

Esterno:

```
<script src="script.js"></script>
```

Javascript può essere inserito nella sezione `<head>`.

Tuttavia è meglio posizionarlo nella parte inferiore della sezione `<body>`, perché posizionare gli script lì migliora la velocità di visualizzazione, poiché la compilazione degli script rallenta la visualizzazione.

Variabili

Utilizzi le variabili come nomi simbolici per i valori nella tua applicazione. I nomi delle variabili, detti identificatori, sono conformi a determinate regole.

Un identificatore JavaScript deve iniziare con una lettera, un trattino basso (`_`) o un segno di dollaro (`$`); i caratteri successivi possono essere anche cifre (`0-9`). Poiché JavaScript fa distinzione tra maiuscole e minuscole, le lettere includono i caratteri da "A" a "Z" (maiuscolo) e i caratteri da "a" a "z" (minuscolo). Tre modi per dichiarare le variabili:

- keyword `var`: può essere usata per dichiarare sia variabili locali che globali
- Assegnando un valore, in questo modo si dichiarerà sempre una variabile globale. Se dichiarata fuori da ogni funzione, genera un strict javascript warning (non si dovrebbe usare)
- keyword `let`: sintassi usata per dichiarare block-scope local variables

- keyword `const`: dichiarare una block-scope local constant

Una variabile utilizzando solo let o var senza specificare il valore assegnato è `undefined`

Lunghezza si una stringa: `.length`. Ogni carattere ha il suo indicem il primo carattere è 0 ed arriva fino a `length-1`

JS data type conversion

È un linguaggio tipizzato dinamicamente. Non è necessario specificare il dato della variabile quando viene dichiarato ed i data types sono convertiti automaticamente come richiesto durante l'esecuzione dello script.

If statement

Se l'esecuzione dell'if e della specifica condizione è vera entra nella prima, altrimenti un altro stato viene eseguito.

Il costrutto if (Expression) Statement costringerà il risultato della valutazione dell'espressione a un valore booleano utilizzando il metodo astratto ToBoolean per il quale la specifica ES5 definisce il seguente algoritmo:

Argomento --> Risultato

Tipo argomento	Risultato
undefined	false
null	false
boolean	risultato uguale all'input (no converison)
number	risultato è falso se l'argomento è +0, -0, NaN; altrimenti true
string	risultato è falso se l'argomento è una stringa vuota (lunghezza zero); altrimenti vero
object	true

Switch

Valuta un espressione, che matcha con il valore della clausola case ed esegue lo stato associato a quel caso

```
switch (expression){
    case value1:
        // ...
        [break;]
    case value2:
        // ...
        [break;]
    ...
    [default:
        // stato eseguito quando nessuno dei valori forniti ha un match
con il valore dell'espressione
        [break;]]
```

```
}
```

Operatori di comparazione

Gli operatori di confronto vengono utilizzati nelle istruzioni logiche per determinare l'uguaglianza o la differenza tra variabili o valori.

Operatore	Descrizione
==	equal to
===	equal value and equal type
!=	not equal
!===	not equal value or not equal type
>	greather than
<	less than
>=	greather than or equal to
<=	less than or equal to

Operatori ternari

L'operatore condizionale (ternario) è l'unico operatore JavaScript che accetta tre operandi. Questo operatore viene spesso utilizzato come scorciatoia per l'istruzione if.

`condition ? expr1 : expr2`

While

L'istruzione while crea un ciclo che esegue un'istruzione specificata fintanto che la condizione di test restituisce true. La condizione viene valutata prima di eseguire l'istruzione.

```
while (condition) {
    ...
}
```

For

L'istruzione for crea un ciclo costituito da tre espressioni facoltative, racchiuse tra parentesi e separate da punto e virgola, seguite da un'istruzione (solitamente un'istruzione di blocco) da eseguire nel ciclo. Per uscire preventivamente usare `break`.

`for ([initialization]; [condition]; [final-expression]) statement`

Può creare un loop che itera tutti gli oggetti che sono dentro un iterabile

```
for (var index in arr){
    console.log(arr[index]);
}
```

```
for (var elem of arr){
    console.log(elem);
}
```

Arrays

È un oggetto globale usato per la costruzione degli array, che sono ad alto livello come gli oggetti.

`.length` puoi accedere con la notazione `arr[arr.length - 1]` ed usare

Funzioni disponibili:

```
    //
});

.push("Orange"); // aggiunge un elemento
.pop(); // remove last element
.shift(); // remove  from the front
.unshift("Strawberry") // add to the front
.push("Mango"); // aggiunge alla fine
indexOf("Banana"); // ritorna l'index dell'elemento. può essere anche un
intero, se non presente ritorna -1
```

Splice

The `splice()` method changes the contents of an array by removing or replacing existing elements and/or adding new elements in place. To access part of an array without modifying it, see `slice()`.

Se applicato ad un array, quest'ultimo viene modificato. Altrimenti ritorna: an array containing the deleted elements.

- If only one element is removed, an array of one element is returned.
- If no elements are removed, an empty array is returned.

Funzioni

La definizione di funzione consiste nell'utilizzo della keyword `function` seguita da:

- Nome della funzione
- Lista dei parametri della funzione, chiusi da parentesi e separati da virgole
- Lo stato JavaScript che definisce la funzione, racchiusa tra parentesi graffe
- È possibile passare come parametro una funzione

Sebbene la dichiarazione di funzione precedente sia sintatticamente un'istruzione, le funzioni possono anche essere create da un'espressione di funzione. Tale funzione può essere anonima; non deve avere un nome.

```
// metodo 1:
function square(number){
    return number * number;
}

// metodo 2:
var square = function(number) { return number * number; };
var x = square(4); // x gets the value 16

// possibile anche definire un nome di una funzione per poter essere
utilizzata all'interno della funzione stessa (o in un debugger per
identificare la funzione in stack)

var factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1); };
console.log(factorial(3));
```

Arrow functions

È un'alternativa sintatticamente compatta a un'espressione di funzione regolare, sebbene senza i bind alle keyword `this`, `arguments`, `super` o `new.target`. Le espressioni di funzione freccia non sono adatte come metodi e non possono essere utilizzate come costruttori.

```
(param1, param2, ..., paramN) => { statements } (param1, param2, ...,
paramN) => expression
// equivalent to: => { return expression; }

// Parentheses are optional when there's only one parameter name:
(singleParam) => { statements }
singleParam => { statements }

// The parameter list for a function with no parameters should be written
with a pair of parentheses.
() => { statements }
```

Prima delle arrow functions, ogni nuova funzione definiva il proprio this

- Un nuovo oggetto nel caso di un costruttore
- `undefined` in `strict` mode function calls
- L'oggetto di base se la funzione è stata chiamata come "metodo dell'oggetto"

In alternativa, è possibile creare una funzione associata in modo che un valore preassegnato venga passato alla funzione di destinazione associata. Una funzione freccia non ha il proprio `this`. Viene utilizzato il valore `this` dell'ambito lessicale di inclusione; le arrow function seguono le normali regole di ricerca delle variabili. Quindi, durante la ricerca di `this` che non è presente nell'ambito corrente, una arrow function finisce per trovare `this` dal suo ambito di inclusione.

Hoisting

L'hoisting è stato pensato come un modo generale di pensare a come funzionano i contesti di esecuzione (in particolare le fasi di creazione ed esecuzione) in JavaScript. Una definizione rigorosa di hoisting suggerisce che le dichiarazioni di variabili e funzioni vengono spostate fisicamente all'inizio del codice, ma in realtà non è ciò che accade. Invece, le dichiarazioni di variabili e funzioni vengono messe in memoria ma rimangono esattamente dove vengono scritte nel codice

```
var stud;
studentName(stud);
 function studentName(name) {
  console.log("My student's name is " + name);
}
stud = "pluto";

print --> My student's name is null
```

Scope

Le variabili hanno scope a livello di funzione, sono visibili nelle funzioni quando sono definite.

Closure

JavaScript consente l'annidamento delle funzioni e garantisce alla funzione interna l'accesso completo a tutte le variabili e funzioni definite all'interno della funzione esterna (e a tutte le altre variabili e funzioni a cui la funzione esterna ha accesso).

```
var func1 = function(name) {
  var getName = function () {
    return name;
  }
  return getName;
}

var pet = func1("Pluto")

console.log(pet()) // returns

// can do also with brackets function

var func2 = function(name) {
```

```
  var getName = () => name;

  return getName;
}

var pet2 = func2("Pluto")

console.log(pet2()) // returns "Pluto"
```

Objects

Un oggetto è una collezione di dati e/o funzionalità. Così come molte cose in JavaScript, creando un oggetto dopo l'inizializzazione di una variabile

`var obj = { [key]:[value][,...]};`

```
var person = {
    name: ["Danilo", "Leo"],
    age: 28,
    bio: function() {
     alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years
old.');
    },
    greeting: function() {
      alert('Hi! I\'m ' + this.name[0] + '.');
    }
};
person.bio()
console.log(person.age)
```

Javascript si basa sull'eredità del prototipo. In Javascript tutto è un Oggetto e ogni Oggetto mantiene un "collegamento" al suo prototipo e questo crea una catena di prototipi in cui gli oggetti possono ereditare comportamenti da altri oggetti. Questo differisce dall'ereditarietà classica in cui si definisce una classe o un design per ogni oggetto e ne si crea un'istanza. È uno dei concetti di basso livello che rende JavaScript un linguaggio multiparadigma molto flessibile.

```
const Animal = {race : "Human"};
const Person = Object.create(Animal);

console.log(Animal.race); //human
console.log(Person.race); //human
```

Bracket notations

Simile al modo in si accede agli elementi in un array: invece di utilizzare un numero di indice per selezionare un elemento, stai usando il nome associato al valore di ciascun membro. Non sorprende che gli oggetti

siano talvolta chiamati array associativi: associano le stringhe ai valori nello stesso modo in cui gli array associano i numeri ai valori.

`person['age']` `person["name"][1]`

This

La parola chiave `this` si riferisce all'oggetto corrente all'interno del quale viene scritto il codice, quindi in questo caso equivale a person. Quando iniziamo a creare costruttori, ecc., questo è molto utile assicurerà sempre che vengano utilizzati i valori corretti quando il contesto di un membro cambia.

Classi

La sintassi della classe non introduce in JavaScript un nuovo modello di ereditarietà orientato agli oggetti. Una principale differenza tra dichiarazione di funzioni e di classi è che le dichiarazioni della funzione è hoisted e quella della classe no. Prima si ha la necessità di dichiarare la classe e successivamente accedere, altrimenti il codice ritorna un `ReferenceError`.

Document Object Model (DOM)

È un'interfaccia di programmazione per documenti HTML. Rappresenta la pagina in modo che i programmi possano modificare la struttura, lo stile e il contenuto del documento. Il DOM rappresenta il documento come nodi e oggetti. In questo modo, i linguaggi di programmazione possono connettersi alla pagina. Ad esempio, il DOM standard specifica che il metodo `getElementsByTagName` nel codice seguente deve restituire un elenco di tutti gli elementi con il relativo nome nel documento: Browser diversi hanno implementazioni diverse del DOM e queste implementazioni mostrano vari gradi di conformità allo standard DOM effettivo, ma ogni browser Web utilizza alcuni modelli di oggetti del documento per rendere accessibili le pagine Web tramite JavaScript. Quando crei uno script, sia esso inline in un elemento `<script>` o incluso nella pagina Web tramite un'istruzione di caricamento dello script, puoi iniziare immediatamente a utilizzare l'API `document` o `window` per manipolare il documento stesso o per ottenere ai figli di quel documento, che sono i vari elementi nella pagina web. Il seguente JavaScript visualizzerà un avviso quando il documento viene caricato (e quando l'intero DOM è disponibile per l'uso):

`<body onload="window.alert('Page loaded!');">`

```
// run this function when the document is loaded
window.onload = function() {
    // create a couple of elements in an otherwise empty HTML page
    var heading = document.createElement("h1");
    var heading_text = document.createTextNode("Big Head!");
    heading.appendChild(heading_text);
    document.body.appendChild(heading);
}
```

Interfaces

Gli oggetti `document` e `window` sono gli oggetti che l'interfaccia generalmente usa più spesso nella programmazione DOM. In parole povere, l'oggetto `window` rappresenta qualcosa come il browser e

l'oggetto `document` è la radice del documento stesso. `element` eredita dall'interfaccia Node generica e insieme queste due interfacce forniscono molti dei metodi e delle proprietà utilizzati sui singoli elementi. Questi elementi possono anche avere interfacce specifiche per gestire il tipo di dati che contengono.

Lista interfacce:

```
document.getElementById(id)
document.getElementsByTagName(name)
document.getElementsByClassName(className) ● document.createElement(name)
document.querySelector(selector)
parentNode.appendChild(node)
element.innerHTML
element.style.left
element.setAttribute()
element.getAttribute()
element.addEventListener()
window.content
window.onload
console.log()
window.scrollTo()
```

Eventi

JavaScript è un linguaggio a thread singolo, il che significa che può eseguire solo un'istruzione alla volta, anche se CPU ha più core e thread.

"In che modo JavaScript gestisce i lavori contemporaneamente?". JavaScript ha un modello di concorrenza basato su un ciclo di eventi. Quando viene eseguito un codice JavaScript, sulla macchina vengono allocate due regioni di memoria, lo stack di chiamate (una memoria ad alte prestazioni) e l'heap. Lo Stack viene utilizzato per eseguire le funzioni e salva una copia frame delle funzioni e una copia delle sue variabili locali. L'heap viene utilizzato in una situazione più complessa. È un pool di memoria non strutturato in cui verranno archiviati elementi come oggetti o valori primitivi all'interno delle chiusure, è Garbage Collected, quindi non si gestisce manualmente l'allocazione della memoria o libera memoria.

Un runtime JavaScript utilizza una coda di messaggi, che è un elenco di messaggi da elaborare. Ogni messaggio ha una funzione associata che viene chiamata per gestire il messaggio. Il runtime inizia a gestire i messaggi in coda, a partire da quello più vecchio. Il messaggio viene rimosso dalla coda e la relativa funzione viene chiamata con il messaggio come parametro di input. Come sempre, la chiamata di una funzione crea un nuovo stack frame per l'uso di quella funzione. L'elaborazione delle funzioni continua finché lo stack non è nuovamente vuoto. Quindi, il ciclo di eventi elaborerà il messaggio successivo nella coda (se presente).

Event Loop si riferisce a una funzionalità implementata dai motori che consentono a JavaScript di scaricare attività su thread separati. Le API del browser e del nodo eseguono attività di lunga durata separatamente dal thread JavaScript principale, quindi accodano una funzione di callback da eseguire sul thread principale al termine dell'attività.

Un event loop può essere usata come una coda di messaggi tra il singolo thread javascript e l'OS

```
while (queue.waitForMessage()) {
    queue.processNextMessage();
}
```

`queue.waitForMessage()` aspetta sincronamente l'arrivo di un messaggio. Ogni messaggio è processato completamente prima che ogni messaggio è processato.

- Ogni messaggio è processato completamente prima che ogni messaggio sia processato
- Se un messaggio impiega troppo tempo per essere complicato, l'applicazione non è capace di processare le interazioni dell'utente come i click o lo scroll. Un browser ritorna il messaggio `a script is taking too long to run`

Una proprietà molto interessante del modello del ciclo di eventi è che JavaScript, a differenza di molti altri linguaggi, non si blocca mai. La gestione dell'I/O viene in genere eseguita tramite eventi e callback, quindi quando l'applicazione è in attesa della restituzione di una query IndexedDB o di una richiesta XHR, può comunque elaborare altre cose come l'input dell'utente. Esistono eccezioni legacy come avviso o XHR sincrono, ma è considerata una buona pratica evitarle.

Una proprietà molto interessante del modello a loop di eventi è che JavaScript, a differenza di molti altri linguaggi, non si blocca mai. La gestione dell'I/O viene in genere eseguita tramite eventi e callback, quindi quando l'applicazione è in attesa della restituzione di una query IndexedDB o di una richiesta XHR, può comunque elaborare altre cose come l'input dell'utente. Esistono eccezioni legacy come avviso o XHR sincrono, ma è considerata una buona pratica evitarle.

Tre modi per registrare degli eventi su un elemento DOM

Usare la funzione `addEventListener` che aggiungerà una funzione ogni volta che lo specifico evento è portato al target

```
myButton.addEventListener('click', greet);
```

Usare l'attributo. Questo modo dovrebbe essere evitato, perché renderebbe il markup più grande e meno leggibile, un bug sarà più difficile da trovare ad esempio

```
<button onclick="alert('Hello world')">
```

Usare l'elemento `event function`. Il problema con questo metodo è che solo un handler può essere settato per elemento ed evento

```
myButton.onclick = function(event){alert('Hello world');};
```

Asincrono

Per permetterci di capire cos'è JavaScript asincrono, dovremmo iniziare da assicurandoci di capire cos'è JavaScript sincrono. Molte delle funzionalità che abbiamo esaminato nei precedenti moduli dell'area di

apprendimento sono sincrone con l'esecuzione del codice e il risultato viene restituito non appena il browser può farlo. Durante l'elaborazione di ogni operazione, non può accadere nient'altro/il rendering viene sospeso. Questo perché, JavaScript è a thread singolo. Può succedere solo una cosa alla volta, su un singolo thread principale, e tutto il resto viene bloccato fino al completamento di un'operazione. Per questi motivi, molte funzionalità dell'API Web ora utilizzano codice asincrono per l'esecuzione, in particolare quelli che accedono o recuperano un qualche tipo di risorsa da un dispositivo esterno, come il recupero di un file dalla rete, l'accesso a un database e la restituzione di dati da esso, l'accesso a un streaming video da una webcam o trasmettere il display a un visore VR. Esistono due tipi principali di stile di codice asincrono che incontrerai nel codice JavaScript, le callback vecchio stile, promise, e le più recente stile async/await.

Asynchronous callbacks

Le callback asincrone sono funzioni specificate come argomenti quando si chiama una funzione che avvierà l'esecuzione del codice in background. Al termine dell'esecuzione del codice in background, chiama la funzione di callback per informare che il lavoro è terminato o per far sapere che è successo qualcosa di interessante. Ad esempio il secondo parametro `addEventListener()` è un esempio di callback asincrona Il primo parametro è il tipo di evento da ascoltare e il secondo parametro è una funzione di callback che viene richiamata quando l'evento viene generato. Quando si passa una funzione di callback come argomento a un'altra funzione, si passa solo il riferimento della funzione come argomento, ovvero la funzione di callback non viene eseguita immediatamente. Viene "richiamato" (da cui il nome) in modo asincrono da qualche parte all'interno del corpo della funzione che lo contiene. La funzione contenitore è responsabile dell'esecuzione della funzione di callback quando arriva il momento.

Asynchronous promises

Le promises sono il nuovo stile di codice asincrono che vedrai utilizzato nelle moderne API Web.

Un promise è un proxy per un valore non necessariamente noto al momento della creazione della promessa. Consente di associare gli handlers all'eventuale valore di successo o al motivo dell'errore di un'azione asincrona. Ciò consente ai metodi asincroni di restituire valori come i metodi sincroni: invece di restituire immediatamente il valore finale, il metodo asincrono restituisce una promessa di fornire il valore in futuro. Una promises è in uno di questi stati:

- in sospeso (pending): stato iniziale, né soddisfatto né rifiutato.
- adempiuto (fullfilled): significa che l'operazione si è conclusa con successo.
- rifiutato (rejected): significa che l'operazione è fallita. Una promessa in sospeso può essere rispettata con un valore o rifiutata con un motivo (errore). Quando si verifica una di queste opzioni, vengono chiamati i gestori associati accodati dal metodo then di una promessa. Se la promise è già stata soddisfatta o rifiutata quando viene collegato un gestore corrispondente, verrà chiamato il gestore, quindi non esiste alcuna condizione di competizione tra il completamento di un'operazione asincrona e il collegamento dei relativi gestori. Poiché i metodi `Promise.prototype.then()` e `Promise.prototype.catch()` restituiscono promesse, possono essere concatenati. I metodi `promise.then()`, `promise.catch()` e `promise.finally()` vengono utilizzati per associare ulteriori azioni a una promessa che viene stabilita.

Il metodo `then()` il metodo richiede fino a due argomenti; il primo argomento è una funzione di callback per il caso risolto della promise e il secondo argomento è una funzione di callback per il caso rifiutato.

Ciascun `.then()` restituisce un oggetto promise appena generato, che può essere utilizzato facoltativamente per il concatenamento.

```
const myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('foo');
    }, 300);
});
myPromise
    .then(handleResolvedB, handleRejectedB)
    .then(handleResolvedA, handleRejectedA)
    .then(handleResolvedC, handleRejectedC);
```

L'elaborazione continua al collegamento successivo della catena anche quando a `.then()` manca una funzione di callback che restituisce un oggetto `Promise`. Pertanto, una catena può tranquillamente omettere ogni funzione di callback di rifiuto fino al `.catch()` finale.

La gestione di una promise rifiutata in ogni `.then()` ha conseguenze più in basso nella catena delle promesse. A volte non c'è scelta, perché un errore deve essere gestito immediatamente. In questi casi dobbiamo lanciare un errore di qualche tipo per mantenere lo stato di errore lungo la catena. D'altra parte, in assenza di una necessità immediata, è più semplice tralasciare la gestione degli errori fino all'istruzione finale `.catch()`. Un `.catch()` è in realtà solo un `.then()` senza uno slot per una funzione di callback per il caso in cui la promessa viene risolta.

```
myPromise
        .then(handleResolvedA)
        .then(handleResolvedB)
        .then(handleResolvedC)
        .catch(handleRejectedAny)
```

La condizione di cessazione di una promessa determina lo stato "regolato" della prossima promessa nella catena. Uno stato "risolto" indica un completamento con successo della promessa, mentre uno stato "rifiutato" indica una mancanza di successo. Il valore di ritorno di ogni promessa risolta nella catena viene passato al successivo .then(), mentre il motivo del rifiuto viene passato alla successiva funzione di gestione del rifiuto nella catena. Le promise di una catena sono annidate come bambole russe, ma vengono fatte aperte come la cima di una pila. La prima promessa della catena è più profondamente nidificata ed è la prima a spuntare.

Async / await

Aggiunte più recenti al linguaggio JavaScript sono le funzioni asincrone e la parola chiave `await`. Queste funzionalità fondamentalmente agiscono come zucchero sintattico in aggiunta alle promesse, rendendo il codice asincrono più facile da scrivere e da leggere in seguito. Rendono il codice asincrono più simile al codice sincrono della vecchia scuola, quindi vale la pena impararli. Prima di tutto abbiamo la parola chiave `async`, che metti davanti a una dichiarazione di funzione per trasformarla in una funzione asincrona. Una

funzione asincrona è una funzione che sa come aspettarsi la possibilità che la parola chiave `await` venga utilizzata per invocare codice asincrono.

```
async function hello() { return "Hello" };
hello();
//or
let hello = async function() { return "Hello" };
hello();
```

Invocare la funzione ora restituisce una promessa. Questo è uno dei tratti delle funzioni asincrone, i loro valori restituiti sono garantiti per essere convertiti in promesse.

Il vantaggio di una funzione asincrona diventa evidente solo quando la si combina con la parola chiave `await` che funziona solo all'interno di funzioni asincrone all'interno del normale codice JavaScript, tuttavia può essere utilizzato da solo con i moduli JavaScript. `await` può essere messo davanti a qualsiasi funzione asincrona basata sulla promessa per mettere in pausa il codice su quella riga fino a quando la promessa non viene soddisfatta, quindi restituire il valore risultante.

```
async function hello() {
    return greeting = await Promise.resolve("Hello");
};

hello().then(alert)
```

Puoi usare `await` quando chiami qualsiasi funzione che restituisce una promessa, incluse le funzioni dell'API Web e all'interno di un'altra funzione asincrona e se necessario gestire gli errori aggiungere try/catch con async/await.

HTTP Request

Sembra un po' come segue: il tuo browser invia una richiesta, attende goffamente che il server risponda alla richiesta e (una volta che il server risponde) elabora la richiesta. Tutta questa comunicazione è resa possibile grazie a qualcosa noto come protocollo HTTP. Normalmente, questo approccio è sincrono.

Ajax

Ajax (JavaScript asincrono e XML) è il modo tradizionale per effettuare una richiesta HTTP asincrona. I dati possono essere inviati utilizzando il metodo HTTP POST e ricevuti utilizzando il metodo HTTP GET.

1. Per effettuare una chiamata HTTP in Ajax, ha la necessità di inizializzare un nuovo metodo `XMLHttpRequest()`, nello specifico l'endopoint dell'URL e il metodo HTTP.
2. Alla fine, usiamo il metodo `open()` insieme al metodo ed all'URL
3. Loggiamo la risposta dell'HTPP sulla console usando `XMLHTTPRequest.onreadystatechange` property che contiene l'event handler da essere chiamato quando l'evento `readystatechanged` è eseguito. On `readystatechange`properties ha due valori, `readyState`e `status`che ci permette

per controllare lo stato della nostra richiesta. Se `readyState` è uguale a 4, significa che la richiesta è fatta. Se la proprietà `readyState` ha 5 risposte.

```
const http = new XMLHttpRequest();
const url = 'https://jsonplaceholder.typicode.com/todos/';
http.open("GET", url);
//define a functions that will be executed when a request is completed
http.onreadystatechange = function(e){
    if (this.readyState == 4 && this.status == 200) {
        console.log(http.responseText)
    }
}
//send the request
http.send();
```

Fetch

L'API Fetch fornisce un'interfaccia JavaScript per l'accesso e la manipolazione delle parti della pipeline HTTP, come richieste e risposte. Fornisce inoltre un metodo globale `fetch()` che fornisce un modo semplice e logico per recuperare le risorse in modo asincrono attraverso la rete. Questo tipo di funzionalità è stato precedentemente ottenuto utilizzando `XMLHttpRequest`. Fetch fornisce un'alternativa migliore che può essere facilmente utilizzata da altre tecnologie come Service Workers. Fornisce anche un'unica posizione logica per definire altri concetti correlati a HTTP come CORS ed estensioni a HTTP.

```
fetch('http://example.com/movies.json')
    .then(response => response.json())
    .then(data => console.log(data));
```

L'uso più semplice di fetch() prende un argomento, il percorso della risorsa che si desidera recuperare e restituisce una promessa contenente la risposta (un oggetto Response).

```
// Example POST method implementation:
const data = { username: 'example' };
fetch('https://example.com/profile', {
    method: 'POST', // or 'PUT'
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify(data),
})
.then(response => response.json())
.then(data => {
    console.log('Success:', data);
})
.catch((error) => {
    console.error('Error:', error);
});
```

La specifica di fetch differisce da altre chiamate http asincrone di seguito modi significativi:

- La promise restituita da fetch() non rifiuterà lo stato di errore HTTP anche se la risposta è un HTTP 404 o 500. Invece, non appena il server risponde con le intestazioni, la promessa si risolverà normalmente (con la proprietà ok della risposta impostata su false se la risposta non è compresa nell'intervallo 200–299) e verrà rifiutata solo in caso di errore di rete o se qualcosa ha impedito il completamento della richiesta.
- `fetch()` non invierà cookie di origine incrociata a meno che non imposti l'opzione di inizializzazione delle credenziali. La specifica ha modificato il criterio delle credenziali predefinito in stesso origine.

JQuery

jQuery è una libreria JavaScript che si concentra sulla semplificazione della manipolazione DOM, chiamate AJAX e gestione degli eventi. Viene utilizzato frequentemente dagli sviluppatori JavaScript. jQuery utilizza un formato, `$(selector).action()` per assegnare uno o più elementi a un evento. Per spiegarlo in dettaglio, $(selector) chiamerà jQuery per selezionare gli elementi del selettore e lo assegnerà a un'API di eventi chiamata `.action()`. Una cosa importante da sapere è che jQuery è solo una libreria JavaScript.

Come funziona ?

Per garantire che il loro codice venga eseguito dopo che il browser ha terminato di caricare il documento, molti programmatori JavaScript racchiudono il loro codice in una funzione di caricamento:

```
window.onload = function() { alert( "welcome" ); };
```

Sfortunatamente, il codice non parte fin quando tutte le immagini non sono state scaricate. Per runnare il codice una volta che il documento è pronto per essere manipolato:

- The jQuery library exposes its methods and properties via two properties of the window object called jQuery and $.
- $ is simply an alias for jQuery and it's often employed because it's shorter and faster to write.

```
$( document ).ready(function() { // Your code here.
});
```

Supporta la maggiorparte dei selettori CSS3

Selectors

Non effettua la cache degli elementi, se viene fatta una soluzione ed è probabile che debba essere fatta un'altra volta, è necessario salvare la selezione in una variabile invece di fare la selezione ripetutamente. Una volta che la selezione è salvata nella variabile, basta chiamare i metodi JQuery sulla variabile. Una selezione prende gli elementi che sono sulla pagina quando la selezione è fatta. Se gli elementi sono

aggiunti alla pagina successivamente, ci sarà modo di ripetere la soluzione o invece aggiungerla alle selezioni salvate nella variabile. Stored selections don't magically update when the DOM changes.

`var divs = $("div");`

```
// Refining selections.
$( "div.foo" ).has( "p" ); // div.foo elements that contain <p> tags
$( "h1" ).not( ".bar" ); // h1 elements that don't have a class of bar
$( "ul li" ).filter( ".current" ); // unordered list items with class of
current $( "ul li" ).first(); // just the first unordered list item
$( "ul li" ).eq( 5 ); // the sixth
```

Manipolazione di elementi

Esistono molti modi per modificare un elemento esistente. Tra le attività più comuni c'è la modifica dell'HTML interno o dell'attributo di un elemento. jQuery offre metodi semplici e cross-browser per questo tipo di manipolazioni. Puoi anche ottenere informazioni sugli elementi usando molti degli stessi metodi nelle loro incarnazioni getter.

```
.html() — Get or set the HTML contents.
.text() — Get or set the text contents; HTML will be stripped.
.attr() — Get or set the value of the provided attribute.
.width() — Get or set the width in pixels of the first element in the
selection as an integer.
.height() — Get or set the height in pixels of the first element in the
selection as an integer. .position() — Get an object with position
information for the first element in the selection, relative to its first
positioned ancestor. This is a getter only.
.val() — Get or set the value of form elements.


// Changing the HTML of an element.
$( "#myDiv p:first" ).html( "New <strong>first</strong> paragraph!" );
```

Sebbene ci siano vari modi per spostare gli elementi all'interno del DOM, ci sono generalmente due approcci: posizionare gli elementi selezionati rispetto a un altro elemento o posizionare un elemento rispetto agli elementi selezionati.

Query provides `.insertAfter()` and `.after()`.

- The `.insertAfter()` method places the selected element(s) after the element provided as an argument.
- The `.after()` method places the element provided as an argument after the selected element.

Several other methods follow this pattern: `.insertBefore()` and `.before()`, `.appendTo()` and `.append()`, and `.prependTo()` and `.prepend()`.

Rimozione di elementi

Esistono due modi per rimuovere elementi dalla pagina: `.remove()` e `.detach()`.

- Utilizzare `.remove()` quando si desidera rimuovere permanentemente la selezione dalla pagina, Sebbene restituisca gli elementi rimossi, a questi elementi non verranno allegati i dati e gli eventi associati se li si restituisce alla pagina.
- Usa `.detach()` se hai bisogno che i dati e gli eventi persistano. Come `.remove()`, restituisce la selezione, ma conserva anche i dati e gli eventi associati alla selezione, in modo da poter ripristinare la selezione nella pagina in un secondo momento. Il metodo `.detach()` è estremamente prezioso se si esegue una manipolazione pesante su un elemento. In tal caso, è utile .detach() l'elemento dalla pagina, lavorarci sopra nel codice, quindi ripristinarlo nella pagina quando hai finito. Ciò limita i costosi "DOM touches" mantenendo i dati e gli eventi dell'elemento.

Creazione di elementi

La sintassi per aggiungere nuovi elementi alla pagina è semplice, quindi si è tentati di dimenticare che c'è un enorme costo di prestazioni per l'aggiunta ripetuta al DOM. Se stai aggiungendo molti elementi allo stesso contenitore, dovrai concatenare tutto l'HTML in un'unica stringa, quindi aggiungere quella stringa al contenitore invece di aggiungere gli elementi uno alla volta. Usa una matrice per riunire tutti i pezzi, quindi uniscili in un'unica stringa per aggiungere:

```
var myItems = [];
var myList = $( "#myList" );
for ( var i = 0; i < 100; i++ ) {
    myItems.push( "<li>item " + i + "</li>" );
}
myList.append( myItems.join( "" ) );
```

Manipolazioni di attributi

Quando viene utilizzata la sintassi della funzione, la funzione riceve due argomenti: l'indice in base zero dell'elemento il cui attributo viene modificato e il valore corrente dell'attributo modificato.

```
// Manipulating a single attribute.
$( "#myDiv a:first" ).attr( "href", "newDestination.html" );

// Manipulating multiple attributes.
$( "#myDiv a:first" ).attr({
    href: "newDestination.html",
    rel: "nofollow"
});
```

Manipolazioni data attributes

Ci sono spesso dati su un elemento che vuoi archiviare con l'elemento. jQuery offre un modo semplice per archiviare i dati relativi a un elemento e gestisce i problemi di memoria per te. Oltre a passare `.data()` una

singola coppia chiave-valore per archiviare i dati, puoi anche passare un oggetto contenente una o più coppie.

```
// Storing and retrieving data related to an element.
$( "#myDiv" ).data( "keyName", { foo: "bar" } );
$( "#myDiv" ).data( "keyName" ); // Returns { foo: "bar" }
```

Traversing

Può essere suddiviso in tre parti fondamentali: genitori, figli e fratelli. jQuery ha un'abbondanza di metodi facili da usare per tutte queste parti. Si noti che ciascuno di questi metodi può essere facoltativamente passato a selettori di stringhe e alcuni possono anche accettare un altro oggetto jQuery per filtrare la selezione.

Parents:

```
$( "span.subchild" ).parent();
// Selecting all the parents of an element that match a given selector:
$( "span.subchild" ).parents( "div.parent" );
$( "span.subchild" ).parents();
// Selecting all the parents of an element up to, but *not including* the
selector: $( "span.subchild" ).parentsUntil( "div.grandparent" );
// Selecting the closest parent, note that only one parent will be
selected
// and that the initial element itself is included in the search:
$( "span.subchild" ).closest( "div" );
// returns [ div.child ] as the selector is also included in the search:
$( "div.child" ).closest( "div" );
```

Il metodo per trovare gli elementi figlio da una selezione include `.children()` e `.find()`. La differenza tra questi metodi risiede nella profondità della struttura del figlio in cui viene effettuata la selezione. `.children()` opera solo su nodi figli diretti, mentre `.find()` può attraversare ricorsivamente i figli, i figli di quei figli e così via.

```
// Selecting an element's direct children:
$( "div.grandparent" ).children( "div" );
// Finding all elements within a selection that match the selector: $(
"div.grandparent" ).find( "div" );
```

Async.js & XML AJAX

L'oggetto XMLHttpRequest fa parte di una tecnologia chiamata Ajax (JavaScript asincrono e XML). Utilizzando Ajax, i dati potrebbero quindi essere passati tra il browser e il server, utilizzando l'API XMLHttpRequest, senza dover ricaricare la pagina web. Le richieste Ajax vengono attivate dal codice JavaScript; il codice invia una richiesta a un URL e, quando riceve una risposta, è possibile attivare una

funzione di callback per gestire la risposta. Poiché la richiesta è asincrona, il resto del codice continua a essere eseguito durante l'elaborazione della richiesta, quindi è fondamentale che venga utilizzata una callback per gestire la risposta.

Sfortunatamente, browser diversi implementano l'API Ajax in modo diverso. In genere ciò significava che gli sviluppatori avrebbero dovuto tenere conto di tutti i diversi browser per garantire che Ajax funzionasse universalmente.

L'asincronia di Ajax coglie alla sprovvista molti nuovi utenti jQuery. Poiché le chiamate Ajax sono asincrone per impostazione predefinita, la risposta non è immediatamente disponibile. Le risposte possono essere gestite solo utilizzando una richiamata. Quindi, ad esempio, il seguente codice non funzionerà:

```
var response;
$.get( "request.php", function( r ) {
    response = r;
});
console.log( response ); // undefined
```

Invece, dobbiamo passare una funzione di callback alla nostra richiesta; questo callback verrà eseguito quando la richiesta ha esito positivo, a quel punto possiamo accedere ai dati che ha restituito, se presenti.

```
$.get( "request.php", function( response ) {
    console.log( response ); // server response
});
```

Il metodo core `$.ajax()` di jQuery è un modo potente e diretto per creare richieste Ajax. Richiede un oggetto di configurazione che contiene tutte le istruzioni necessarie a jQuery per completare la richiesta. Il metodo `$.ajax()` è particolarmente prezioso perché offre la possibilità di specificare callback sia di successo che di errore. Inoltre, la sua capacità di prendere un oggetto di configurazione che può essere definito separatamente semplifica la scrittura di codice riutilizzabile.

```
// Using the core $.ajax() method
$.ajax({
    // The URL for the request
url: "post.php",
// The data to send (will be converted to a query string) data: {
id: 123 },
    // Whether this is a POST or GET request
type: "GET",
// The type of data we expect back dataType : "json",
})
// Code to run if the request succeeds (is done); // The response is
passed to the function .done(function( json ) {
     $( "<h1>" ).text( json.title ).appendTo( "body" );
     $( "<div class=\"content\">").html( json.html ).appendTo( "body" );
  })
// Code to run if the request fails; the raw request and // status codes
```

```
are passed to the function .fail(function( xhr, status, errorThrown ) {
    alert( "Sorry, there was a problem!" );
    console.log( "Error: " + errorThrown );
    console.log( "Status: " + status );
})
  // Code to run regardless of success or failure;
.always(function( xhr, status ) { alert( "The request is complete!" );
});
```

Le funzioni di convenienza Ajax fornite da jQuery possono essere modi utili e concisi per soddisfare le richieste Ajax. Questi metodi sono solo "wrapper" attorno al metodo principale `$.ajax()` e semplicemente preimpostano alcune delle opzioni sul metodo `$.ajax()`.

```
// Using jQuery's Ajax convenience methods // Get plain text or HTML (or
POST) $.get( "/users.php", {
userId: 1234
}, function( resp ) {
console.log( resp ); // server response });
 // Add a script to the page, then run a function defined in it
$.getScript( "/static/js/myScript.js", function() {
functionFromMyScript();
});
 // Get JSON-formatted data from the server
$.getJSON( "/details.php", function( resp ) { // Log each key in the
response data $.each( resp, function( key, value ) {
        console.log( key + " : " + value );
    });
});
```

Typescript

È un superset di Javascript. TypeScript si basa su JavaScript. Per prima cosa, scrivi il codice TypeScript. Quindi, compili il codice TypeScript in codice JavaScript semplice utilizzando un compilatore TypeScript. Una volta che hai il codice JavaScript semplice, puoi distribuirlo in qualsiasi ambiente in cui viene eseguito JavaScript.

TypeScript utilizza le sintassi JavaScript e aggiunge sintassi aggiuntive per supportare i tipi. Se hai un programma JavaScript che non presenta errori di sintassi, è anche un programma TypeScript. Significa che tutti i programmi Js sono programmi TypeScript.

- Supporta le altre librerie di Js
- Js è Ts
- Ts è portable, può essere runnato in ogni ambiente dove è possibile a sua volta runnare Js
- È più leggibile
- È più esplicito
- È compilato

Type

Typescript usa le type annotation per specificare esplicitamente i tipi per gli identificatori come variabili, funzioni, oggetti, etc. Typescript usa la sintassi `:type`dopo un identificatore come type annotation che può essere qualsiasi tipo valido. Una volta che l'identificatore è annotato con un tipo, può essere usato solo da quel tipo (altrimenti il compilatore ritornerà un errore)

```
let variableName: type;
let variableName: type = value;
const constantName: type = value;
```

In questa sintassi, l'annotazione dei tipi viene dopo il nome della variabile (o costante) ed è preceduto da `:`.

Primitives

```
let name: string = 'John';
let age: number = 25;
let active: boolean = true;
```

Arrays

```
let arrayName: type[];
let names: string[] = ['Pippo', 'Pluto'];
```

Objects

```
type Person = {
  name: string;
  age: number;
};
```

Function annotation

Nel seguente esempio è possibile assegnare ogni funzione che accetta una stringa e ritorna la stringa.

```
let greeting : (name: string) => string;
greeting = function (name: string) {
    return name;
};
```

Type inference

```
let counter = 0;
// It is equivalent to the following:
let counter: number = 0;
```

Come con le variabili, typescript inferisce il tipo al parametro alla variable di ritorno al tipo del valore di default.

```
function increment(counter: number){
    return counter++;
}

//It is same as:

function increment(counter: number) => number {
    return counter++;
}
```

Considerando un array:

`let items = [1, 2, 3, null]`

Per inferire l'array di items, typescript ha la necessità di considerare il tipo di ogni elemento nell'array. Utilizza il **best common type algorithm** per analizzare ogni tipo e seleziona il tipo compatibile con tutti i candidati. Nel caso sopra, `number[]`

Tipi primitivi

Tutti i numeri in typescript sono floating-point o big integers. I numeri floating-points hanno il tipo number mentre gli interi bigint

- `number`
- `bigint`
- `string`
- `boolean`
- `null`
- `undefined`
- `symbol`

Typescript ha un tipo chiamato `Object`. Il tipo `object` rappresenta tutti i valori non primitivi mentre il tipo `Object` descrive la funzionalità di tutti gli oggetti.

Per esempio, il tipo `Object` ha `toString()` e `valueOf()` i metodi che possono essere accessibili ad ogni oggetto. Ts ha un altro tipo chiamato `empty type` denotato da `{}` che è abbastanza simile al tipo oggetto. L' `empty type` descrive un oggetto che non ha proprietà. Se si prova ad accedere alle proprietà dell'oggetto, typescript ritonerà un compile-time error.

Type tuple

Una tupla lavora come un array lavora con alcune considerazioni addizionali.

- Il numero degli elementi in una tupla è fisso
- I tipi degli elementi sono conosciuti e non hanno bisogno di essere gli stessi

Per esempio, è possibile usare una tupla per rappresentare un valore come coppia di `string` e `number`. L'ordine dei valori in una tupla è importante in quanto se si cambiasse l'ordine si otterrebbe un errore. Per questa ragione, è una buona pratica usare le tuple con i dati che sono relative ad ogni altra in un ordine specifico.

Type enum

Un enum è un gruppo di valori constanti. Sta per **enumerated type**, per definire un enum:

- Keyword enum
- Definire valori costanti dell'enum

```
enum name {constant1, constant2, ... }
```

È buona pratica usare i valori costanti definiti dagli enums nel codice. Definisce il valore numerico di membri enum basati sull'ordine dei membri che appaiono nella definizione di enum, è possibile esplicitare questi numeri per i membri dell'enum

```
enum Month {
    Jan = 1,
    Feb,
    Mar,
    ...
}
```

Type any

Il tipo `any` permette di assegnare un qualsiasi valore ad ogni variabile. Se si dichiara una variabile senza specificare il valore, typescript assegnerà il valore any.

Type void

Tipicamente usato quando una funzione, ad esempio, non ritorna un valore

```
function log(message): void {
    console.log(message);
}
```

È buona pratica aggiungere il tipo void come ritorno di una funzione o un metodo che non ritorna alcun valore. In particolare:

- si migliora la chiarezza del codice
- assicura che sia type-safe: non verrà mai assegnata la funzione con valore di ritorno voi ad una variabile. Nota: se si utilizza il tipo void per una variabile, è possibile assegnare solo il valore undefined.

Type never

Il tipo never è un tipo che non contiene valori. Per via di ciò, non si può assegnare alcun valore con il tipo never. TTipicamente viene utilizzato per rappresentare il valore di ritorno di una funzione che ritorna sempre un errore

```
function raiseError(message: string): never {
    throw new Error(message);
}
```

Le variabili possono anche acquisire il tipo `never` quando si restringe il tipo del tipo che non può essere mai vero.

Type union

Descrive un valore che può essere di diversi tipi. Ad esempio

```
function add(a: number | string, b: number | string){
    if (typeof a === 'number' && typeof b === 'number'){
        return a + b;
    }
    if (typeof a === 'string' && typeof b === 'string') {
        return a.concat(b)
    }
    throw new Error("Error")
}
```

Aliases

Permettono di creare dei nuovi nomi per tipi esistenti.

```
type alias = existingType;
```

È utile per creare tipi di alias per union types, ad esempio

```
type alphanumeric = string | number; let input: aplhanumeric; input = 100; //
valid input = 'Hi'; // valid input = false; // compiler error
```

Type literal string

String literal types permette di definire un tipo che accetta solo una specifica string literal.

```
let click: 'click';
```

Il `click`è una string literal type che accettano solo la string literal `click`. Se si assegna la stringa click sarà valido, tuttavia quendo si assegna un altro string literal ed il click ed il compilatore ritorna errore. String literal type è utile per limitare i possibili valori della stringa. Possono essere combinati con gli union type e egli alias che definiscono un finito set di string literal values per una variabile.

Parametri opzionali

Javascript supporta i parametri opzionali di default. Il compilatore controlla:

- Il numero degli argomenti è diverso dal numero di parametri specificati nella funzione
- Il tipo degli argomenti non sono compatibili con quelli dati come la funzione dei parametri

Per fare si che il parametro sia opzionale basta anteporre `?`.

Rest params

Un parametro rest permette che una funzione accetti zero o più argomenti sul tipo specifico. Nel typescript, seguono le seguenti regole:

- Una funzione ha solo un rest parameter
- Il parametro rest appaie alla fine nella lista dei parametri
- Il tipo del rest parameter è di tipo array

Per dichiararlo è necessario anteporre al nome del parametro tre punti ed usare l'array type annotation

```
function fn(...rest: type[]){
    //...
}
```

Overload di funzioni

Il numero di parametri richiesti deve essere lo stesso. Se un overload ha più parametri di un altro, allora si deve inserire il terzo parametro come opzionale.

(esempio)

Classi

Javascript non ha un concetto di classe come gli altri linguaggi di programmazione. ES6 permette di definire una classe come semplice zucchero sintattico per creare un costruttore ed una funzione. Typescript aggiunge le type annotation e le proprietà sui metodi e le classi

```
class Person {
    firstName: string;
    year: number;

    constructor(firstName: string, year: number){
```

```
        this.firstName = firstName;
        this.year = year;
    }

    hello(): string {
        return this.firstName;
    }
}
```

Access modifier

Possono cambiare la visibilità delle properties e dei metodi di una classe. Nota che TypeScript controlla logicamente l'accesso durante la compilazione, non in fase di esecuzione.

- `private` il modificatore limita la visibilità solo alla stessa classe. Quando si aggiunge questo modificatore a una proprietà o metodo, è possibile accedere a tale proprietà o metodo all'interno della stessa classe. Qualsiasi tentativo di accedere a proprietà o metodi privati al di fuori della classe farà causare un errore in fase di compilazione.
- `public` consente a tutti di accedere alle proprietà e ai metodi della classe posizioni. Se non specifichi alcun modificatore di accesso per proprietà e metodi, lo faranno prendere il modificatore pubblico per impostazione predefinita.
- `protected` modifier consente alle proprietà e ai metodi di una classe di essere accessibili all'interno stessa classe e all'interno di sottoclassi. Quando una classe (classe figlia) eredita da un'altra classe (classe genitore), è una sottoclasse di classe genitore.

Typescript ha introdotto `readonly` modificatore che permette di contrassegnare le proprietà di una classe immutabile. L'incarico ad a sola lettura la proprietà può verificarsi solo in uno dei due posti, nella dichiarazione di proprietà e nel costruttore della stessa classe.

Getter e setter

Permettono di controllare l'accesso a properties della classe:

- un metodo getter ritorna il valore della poperties. Un getter è detto anche accessor
- Un metodo setter aggiorna il valore delle properties, chiamato anche mutator

Ereditarietà

La classe eredita properties e metodi è chiamata classe figlio, mentre la classe i cui metodi e proprietà sono ereditate è detta parent.

Per ereditare una classe basta utilizzare la kewyord `extends`. Per chiamare il costruttore della classe parent dentro la classe child basta usare `super()`. I metodi vanno richiamati con `super.nomeMetodo()`

Static

Una proprietà statica è condivisa tra tutte le istanze della classe. Per dichiarare una proprietà static, è necessario usare la kewyord `static`. Per accedere ad una proprietà statica basta utilizzare `className.propertyName`. Il metodo statico è anche condiviso tra le istanze della classe

Abstract

Tipicamente usato per definire comportamenti comuni per le classi derivate da estendere. Una classe astratta non può essere istanziata direttamente, per utilizzarla è necessario usare la keyword `abstract`. Tipicamente una classe astratta contiene uno o più metodi astratti che non contengono implementazione. Definisce solo la firma del metodo senza implementare il metodo.

Interface

Permettono di descrivere tipi di funzioni. Per descrivere un tipo di funzione, è necessario assegnare l'interfaccia alla signare function che contiene la lista di parametri con i tipi ed i valori di ritorno.

```
interface StringFormat {
    (str: string, isUpper: boolean): string
}

let format: StringFormat;

format = function (str: string, isUpper: boolean){
    return isUpper ? str.toLocaleUpperCase() : str.toLocaleLowerCase();
}

console.log(format('hi', true));
```

Serve per definire un contratto tra classi non collegate. Per estendere un'interfaccia basta usare `extends`. Un'interfaccia può estendere interfacce multiple, creando una combinazione tra tutte le interfacce. L'interfaccia eredita le proprietà ed i metodi della classe, può anche ereditare i membri protetti e private della classe, non solo i pubblici. Quindi l'interfaccia può essere implementata solo da quella classe o sottoclassi di quella classe da cui si estende l'interfaccia.

Type casting

Non ha concetti di type casting perche le variabili hanno tipi dinamici. È comunque possibile convertire una variabile da tipi ad un altro. È possibile usare la keyword `as` o l'operatore `<>`. Inoltre, la parola chiave o l'operatore di casting possono essere utilizzati per l'asserzione del tipo o il restringimento del tipo.

```
let input = document.querySelect('input["type=text"]');
let enteredText = ( input as HTMLInputElement).value;

let input = <HTMLInputElement>document.querySelector('...');
```

Frontend

Nel front-end risiedono tutte le cose visualizzabbili sulla web application. Per esempio, le immagini i bottoni e tutti gli altri elementi grafici e le interazioni con l'utente finale. Tecnologie:

- HTML: linguaggio di markup utilizzato per definire la struttura del documento HTML
- CSS: cascade language utilizzato per definire regole, stile e behaviors di elementi htm
- JS & TS: linguaggio di programmazione usati per l'iterazione con l'utente e la applicaiton logic

Utilizzare il DOM può essere ripetitivo e difficile da gestire, per semplificare ciò basta utilizzare un framework Javascript. In questo modo è possibile evitare di gestire il DOM direttamente:

- React.js: JS & TS, più di una libreria, minimale dal design
- Angular: TS, piattaforma completa, non molto flessibile
- Vue.js: TS & TS, progressive framework, sistema a plugin
- Svelte.js: compila il javascript, no virtual DOM, minimalista e flessibile
- Solid.js: compila JS, no virtual DOM, alte performance
- Alpine.js: usa JS con l'HTML, estende l'html, leggero

Tutti i framework di FE, Javascript e Typescript non hanno molte features di tutti i linguaggi di programmazione, come ad esempio l'abilità di divedere il codice in diversi file ed organizzare il codice. Per risolvere questo problema bisogna utilizzare un budler che organizza un sett di file sorgenti in un singolo file

- Webpack: long-term caching mecanism, code splitting and lazy loading, handles the dependecies automatically
- Rollup: aiuta l'ottimizzazione di applicazioni, supporta tree-shaking e scope-hoisting, offre più elasticità
- Snowpack: builda le dipendenze una volta sola, alte performance, supporta lazy loading

CSS come linguaggio ha gli stessi problemi di js. È difficile organizzare il docie in file diversi ed a CSS mancano molte feature utili. Successivamente possiamo usare il bundler per il CSS, chiamati preprocessors:

- SASS: organizza il css in diversi file, migliora il CSS, logica condizionale
- Less: libreria js, non ha logica condizionale, offre funzioni già create
- Stylus: combinazione di SASS e Less, logica condizionale e postfix conditional, support all mixins.

I preprocessors CSS e JS sono anche utilizzati per sviluppare un set di stili, funzionalità e widget pronti per essere utilizzati in una web application. Questi sono chiamati framework CSS, sono di base un insieme di codici JS e CSS che risolvono problemi comuni e permettono di salvare molto tempo

- Bootstrap: più popolare, fully-feature, mature and customizable
- Foundation: mobile first, generic style, email design and animations
- Bulma: aestethic design, easy to customize, no js
- Tailwind: atomic css, reusable components, no design

Backend

Ogni computer che è connesso ad internet può inviare messaggi attraverso internet ad un altro computer. Il computer che invia il messaggio è chiamato Client ed il computer che lo riceve è detto Server. Prima che ciò accada, il computer non può ricevere messaggi di default, bisogna che sia in grado di ricevere messaggi. La maggiorparte dei linguaggi di programmazione ha delle caratteristiche per fare diventare un computer un server e permettere di ricevere messaggi. Tra gli aspetti legati a reti di computer, vi sono anche le porte socket. Nonostante il supporto di molti linguaggi di programmazione, è difficile scrivere dei back-end da zero, di conseguenza si utilizzano vari framework.

I packages, sono di solito liberie per semplificare lo sviluppo che prendono il vantaggio di soluzioni già scelte, come performare calcoli matematici, connessioni a database o gestire autenticazioni di utenti. Per installare questi packages si utilizzando dei package manager (npm per js, maven per java, composer per php ...)

Un database aiuta a salvare e gestire i dati, è solo un pezzo di software che solitamente è runnato in un computer diverso e bisogna effettuare dei setup nel backend per comunicare con il database.

- Mysql: database relazione, singolo processo, made for high volumes of read
- PostfreSQL: object relational db, multi process, for high volume I/O
- MongoDB: JSON document db, distribuito, per dati non strutturati

La lista di diversi tipi di richieste permesse nel backend è chiamata API (Application Programming interface) e sono uno dei concetti più importanti nel be. QUesti permettono agli altri linguaggi di comunicaer tra loro.

```
app.post("/orders", (request, response) => { const order = new
Order(request); database.save(order); response.send("Order saved"); })

app.get("/orders", (request, response) => { const orders =
database.getOrders(); response.json(orders); })

app.delete("/orders/:id", (request, response) => {
database.deleteOrder(request); response.send("Order deleted"); })
```

Per convenzione vengono chiamate REST (REpresentational State Transfer). In REST il tipo di richiesta ha un significato speciale. Un API che usa le REST naming convention è chiamata REST API, non è un protocollo ma una architettura.

Infrastruttura

Al giorno d'oggi, invece di acquistare i propri computer per eseguire le proprie applicazioni web, le aziende noleggiano computer da una società di cloud computing.

L'idea di base del cloud computing è che stai noleggiando un sacco di computer. Questo è anche noto come IaaS (Infrastructure as a Service). Dietro le quinte, queste aziende hanno un computer gigante e potente e all'interno del suo software sono in esecuzione molti computer più piccoli e stiamo noleggiando uno di questi computer più piccoli. Questi computer più piccoli esistono solo nel software, quindi chiamiamo il tema Macchine virtuali (o vms). Tipicamente, questi sono gestiti da un hypervisor (suchash kvm o qemu) tramite alcuni strumenti (VMWare, Proxmox, Openstack).

Quindi, per eseguire un'applicazione Web, affittiamo una macchina virtuale da una società cloud per eseguire il nostro back-end e affittiamo anche un'altra macchina virtuale per eseguire il nostro database.

Un altro problema che dobbiamo risolvere è che cosa succede se la nostra applicazione web diventa molto popolare durante un certo periodo e iniziamo a ricevere molte richieste e traffico Internet che il nostro server non è in grado di gestire?

Con il cloud computing, possiamo configurare più VM che eseguono lo stesso software di back-end, quindi configurare una VM speciale davanti a queste denominata Load Balancer. Un Load Balancer ridistribuisce le richieste in modo uniforme tra le nostre macchine virtuali. Una volta terminato il periodo di traffico intenso, possiamo semplicemente spegnere alcune VM quando non ne abbiamo bisogno.

Abbiamo ancora un altro problema. Ora abbiamo molte macchine virtuali che dobbiamo creare e configurare e questo richiede molto tempo e fatica. Le società di cloud computing offrono un altro servizio chiamato PaaS (Platform as a Service). Questo ci consente di caricare il nostro codice di back-end e imposterà tutte le VM e i bilanciatori di carico per noi.

Nel mondo reale un back-end può essere costituito da milioni di righe di codice, quindi ci dividiamo in tre basi di codice. Quindi ciascuna di queste basi di codice avrà il proprio back-end, il servizio di bilanciamento del carico e il database.

Quando abbiamo bisogno di inviare un'e-mail, il nostro back-end degli ordini invierà una richiesta al back-end dell'e-mail, che invierà l'e-mail. La suddivisione del nostro back-end in back-end separati è chiamata Microservizi. Ci aiuta a mantenere il nostro codice di base più piccolo e mirato.

Ogni microservizio non deve utilizzare lo stesso sistema di bilanciamento del carico, linguaggio di programmazione e database. Ad esempio, un microservizio può utilizzare Javascript e MongoDB, un altro può utilizzare Python e PostgreSQL. Inoltre, un Microservizio potrebbe essere offerto da un'altra azienda.

Quando un'azienda fornisce un backend e un'API che possono essere utilizzati da applicazioni esterne, questo viene chiamato SaaS (Software as a Service)

Cloud:

- SaaS: utente finale (Office365, Trello, SalesForce)
- PaaS: application developers (SAP cloud platform, google app engine, AWS elastic beanstalk..)
- IaaS: microsoft azure, google compute engine, AWS .

Al giorno d'oggi la maggior parte delle aziende utilizza il cloud computing per eseguire il back-end delle proprie applicazioni Web invece di acquistare e gestire i server fisici (Bare Metal).

In precedenza abbiamo introdotto alcuni database come MySQL, PostgreSQL e MongoDB. Questi a volte sono chiamati Database Primari, perché sono i database principali utilizzati dalla nostra applicazione web.

Generalmente iniziamo il nostro back-end con un server e un database primario e poi inseriamo queste tecnologie aggiuntive se necessario. Se consentiamo ai nostri utenti di caricare immagini, un database primario non è adatto per archiviare tali dati, quindi potremmo utilizzare uno store BLOB come un comune servizio CDN per archiviare e caricare le immagini caricate dagli utenti.

Se la nostra applicazione Web sta ricevendo molto traffico e dobbiamo scaricare un po' di stress dal nostro database principale, aggiungeremo un servizio di cache, come Redis, per migliorare le prestazioni. Inoltre,

se abbiamo bisogno di pianificare qualcosa o una coda di lavoro, useremmo RabbitMQ per pianificare alcune attività o messaggi.

Nodejs

Node.js è un ambiente per eseguire Javascript al di fuori del browser e quindi al di fuori della sandbox del browser. È stato creato nel 2009 ed è basato su Chrome V8 JS Engine. Con l'aiuto di Node non è mai stato così facile creare applicazioni Web Full Stack, il frontend e il backend sono costruiti utilizzando la stessa lingua!

È open source e multipiattaforma ed è perfetto per applicazioni ad alta intensità di dati e in tempo reale.

Browser

- DOM
- Window
- Interactive App
- No Filesystem
- Fragmentation
- ES6 Modules
- Browser Console

Node.js

- No DOM
- Global
- Server Side Apps
- Filesystem
- Versioning
- CommonJS
- O.S. Terminal

Per runnare un codice JS: `node <filename>`sul terminale. Tutte le vanilla functions trovate sull'oggetto window sul web browser sono trovate nel progetto global. In Node.js ci sono diverse variabilio globali. Nel browser sappiamo che abbiammo accesso all'oggetto window e possiamo ottenere un sacco di cose utili da questo globale, ad esempio un querySelector o un fetch. Non c'è oggetto finestra. In Node.js esiste un vero concetto di variabili globali.

- `_dirname`: path della directory corrente
- `filename`: filename corrente
- `require`: funzione che usa i moduli
- `module`: informazioni riguardo il modulo current modules
- `process`: informazioni riguardo l'env, quando il programma è eseguito

Quando si esegue un'applicazione Node.js si esegue un file, ma l'applicazione può essere un insieme di file. Il primo file è comunemente noto come Entrypoint o File principale. Normalmente, il codice dell'applicazione gigante verrà suddiviso in moduli. Questi sono simili ai moduli JS e React vanilla ma la sintassi è diversa, perché Node.js usa CommonJs per caricare i moduli. Di di default, ogni file in node è un modulo!

Quindi per creare un nuovo modulo, devi creare un nuovo file vuoto.

Ad esempio, creare un nuovo file in cui definiremo alcune variabili, una privata (forse un token segreto) e una variabile pubblica che vogliamo esportare. Nelle variabili globali abbiamo il modulo global che stampa alcune informazioni sul modulo. C'è un oggetto chiamato exports che definisce cosa verrà esportato dal modulo. Quindi possiamo decidere cosa può essere “pubblico” e cosa non lo è.

Infine, possiamo importare il nuovo modulo con la funzione require global, usando la dot path syntax.

```
//names.js file
// local
const SECRET = "token";
// public
const name = "Danilo";

module.exports = { name }

//myconsole.js

const trace = (message) => {
    console.log(message);
}
module.exports = trace

//app.js file

const names = require("./names.js");
const trace = require("./myconsole.js");
trace(names);
```

Built-in modules

Node.js ha molti moduli built-in. Per caricare uno di questi devi usare il loro nome invece del path nella funzione `require`.

OS module

Può utilizzare molte informazioni riguardo il sistema operativo e più.

```
const os = require("os");

const user = os.userInfo();
console.log(user);

console.log(os.uptime())

const currentOS = {
    name: os.type(),
    release: os.release(),
```

```
    totalMem: os.totalmem(),
    freeMem: os.freemem()
}

console.log(currentOS);
```

Path modules

Il modulo permette di interagire con i path / filesystem dei sistemi oeprativi

```
const path = require("path");

console.log(path.sep) // platform specific separator

// joins sequence of path using os separator
const filePath = path.join("/path", "directoryName", "fine.txt");

const base = path.basename(filePath) // get basename

const absolute = path.resolve(_dirname, "contentdir", "file.txt"); //
resolve to get absolute path
```

FS module

Permettono di interagire con il filesystem del so. Può lavorare in modo sincrono ed asincrono.

Sincrono:

```
const {readFileSync, writeFileSync} = require("fs");
// same of const fs = require("fs");
// fs.readFileSync()

const first = readFileSync("./file.txt");
const second = readFileSync("./file2.txt");

console.log(first, second);

// concatenate two file into new one (or append)

writeFileSync(
    "./file3.txt", // path
    first + second, // content
    {   // options
        flag: "a" // append
    }
);
```

Asincrono:

```
readFile("./file.txt", "utf-8", (err, result) => {
    if (err){
        console.log(err);
        return;
    }
    const first = result;
    readFile("./file2.txt", "utf-8", (err, result) => {
        if (err){
            console.log(err);
            return;
        }
        const second = result;
        writeFile("./file3.txt", first+second);
    });
});
```

Alcune operazioni possono prendere molto tempo per essere elaborate, bloccando altri task. L'idea di Node.js e le operazioni sincrone è quella di gestire la concorrenza in maniera asincrona. Vi sono anche le promises e i metodi async/await.

HTTP

Permette di instanziare un server o client http.

```
const http = require("http");

// run a basic HTTP server, non vi è URL nella logica REST

const server = http.createServer((req, res) => {
    res.write("Response");
    res.end();
})

// listen to TCP port 8080
server.listen(8080)

// esempio di routing REST

const server = http.createServer((req, res) => {
    if (req.url === "/") {
        res.end("Home page")
    }
    if (req.url === "/about"){
        res.end("About")
    }
    res.end("<h1>404</h1>");
```

```
})
```

NPM

Node Package Manager è il node package manager di node.js grazie al quale è possibile caricare e condividere il codice. I pacchetti sono moduli esterni con le loro dipendenze. Non vi è controllo qualità, è possibile trovare codice non funzionante o malware

Install locally the package and its dependencies

```
npm <i | install> <package name>
```

Install globally the package and its dependencies

```
npm <i | install> -g <package name>
```

Uninstall

```
npm <u | uninstall> <package name>
```

Initialize the manifest of the project (package.json che conserva informazioni importanti riguardo il progetto ed i packages)

```
npm init [-y]
```

Package.json

File più importante che definisce alcune informazioni importanti riguardo al progetto:

- Nome, descrizione e versione del progetto
- Nome dell'autore
- Alcuni script che possono essere personalizzati
- Main del progetto
- Lista delle dipendenze e la loro versione
- Lista delle dipendenze di sviluppo e la loro versione

Permettono di committare e deployare il progetto senza le dipendenze installate, può essere installate anche dopo la prima run.

Quando un pacchetto viene installato (ad esempio localmente), nel tuo progetto viene creata automaticamente una cartella di progetto denominata `node_modules` e i pacchetti vengono installati in questa cartella.

Infine, quando abbiamo bisogno di caricare il nome di un pacchetto (come bootstrap) possiamo usare solo il nome del pacchetto (`require("bootstrap")`).

Questo meccanismo permette di differenziarsi da moduli personali e moduli di terze parti. Inoltre, un pacchetto può richiedere molte dipendenze e questa cartella potrebbe diventare davvero pesante in termini di spazio su disco.

Quindi, quando il progetto è pronto per il commit e la distribuzione, è sufficiente caricare l'intero progetto senza la cartella `node_modules` e avviare l'installazione `npm` per installare tutte le dipendenze definite nel file `package.json` (quindi aggiungere `node_modules` nel proprio `.gitignore!`).

Events

In `Node.js`è possibile ascoltare alcuni eventi specifici e successivamente runnare callback sul trigger. È possibile definire eventi personalizzati. Per gestire ed utilizzare eventi è necessario usare dei moduli built.in . Per gestire questi eventi possiamo usare `EventEmitter` dai built-in module `events`.

```
const EventEmitteer = require("events");

// custom event emitter
const customEmitter = new EventEmitter();

// listen specific event
customEmitter.on("response", (name, age) => {
    console.log("data received " + name + " " + age);
})

// can have multiple callbacks for the same event
customEmitter.on("response", () => {
    console.log("other logic")
});

// the order MATTER

// emit specific event

customEmitter.emit("response", "Mario", 35);
```

Http

I moduli http creano un server che è un'istanza EventEmitter e ha i suoi eventi, come l'evento di richiesta che viene attivato quando un utente fa una richiesta al nostro server.

```
const http = require("http");
const server = http.createServer();

server.on("request", (req, res) => {
    res.end("Welcome");
});

server.listen(8080);
```

Streams

Sono usati per leggere o scrivere in sequenza. Quando dobbiamo gestire e manipolare i dati in streaming, ad esempio il sorgente continuo o un file di grandi dimensioni, gli stream sono molto utili ed estendono la classe EventEmitter. Quattro tipi:

- Writeable: usati per scrivere dati sequenzialmente
- Readable: usati per leggere i dati sequenzialmente
- Duplex: usati per leggere e scrivere dati sequenzialmente
- Transform: i dati possono essere modificati quando scrivi o leggi

Sono utili e gestiscono anche grandi file da inviare in risposta in http:

```
const {createReadStream} = require("fs");
const stream = createReadStream("./bigfile");

stream.on("data", (result) => {
    //buffer size is 64kb, can be controlled with options
    console.log(result);
});

stream.on("error", (err) => {
    console.log(err);
});

const http = require("http");
const fs = require("fs");

http.createServer((req, res) => {
    const fileStream = fs.createReadStream("./bigfile", "utf-8");
        fileStream.on("open", () => {
            fileStream.pipe(res);
        });
        fileStream.on("error", (err) => {res.end(err)});
}).listen(8080);
```

Express.js

È un framework di Node.js minimale e flessibile creato per sviluppare web app ed API in modo veloce e semplice. Express è costruito sul modulo http ma non è un modulo integrato ma al giorno d'oggi è uno standard per la creazione di applicazioni web.

Rende molto più semplice la creazione di applicazioni Web con Node.js Utilizzato per app server, servizi API e microservizi Estremamente leggero, veloce e gratuito Controllo completo di richieste e risposte Il framework più popolare su Node.js Ottimo da utilizzare con framework lato client

- Rende molto più semplice la creazione di applicazioni Web con Node.js
- Utilizzato per app server, servizi API e microservizi
- Estremamente leggero, veloce e gratuito

- Controllo completo di richieste e risposte
- Il framework più popolare su Node.js
- Ottimo da utilizzare con framework lato client

Per quanto riguarda il modulo http, è necessario creare un'istanza dell'applicazione express che abbia lo stesso metodo listen per eseguire il server http su una porta specifica.

```
const express = require("express");
const app = express();

app.get("/", (req, res) => {
    res.send("Home page");
})

app.listen(8080, () => {
    console.log("Server listen on port 8080");
})
```

App methods

- get: GET request management
- post: POST request management
- put: PUT request management
- all: to manage all type of request
- use: this is responsible for middleware

Static files

Negli esempi precedenti restituiamo semplicemente una stringa, una stringa HTML o un singolo file al client quando questo fa una richiesta GET (per esempio alla root route). Nella realtà, un'applicazione web (frontend) ha molti file, come HTML, CSS, JS, immagini, ecc. file che sono difficili da gestire uno per uno. Inoltre, questi file non cambiano nel tempo ma solo se lo sviluppatore aggiornerà il codice frontend. Infatti, l'applicazione frontend può aggiornarsi autonomamente tramite javascript, quindi può modificare l'interfaccia utente e le informazioni con lo stesso codice senza generarlo dinamicamente. Pertanto, possiamo dire che, un tipico codice frontend e tutti i suoi asset sono file “statici”

I file statici sono in genere file che il server non deve modificare o generare. Per definirlo utilizzeremo il middleware (lo vedremo più avanti). Express gestirà tutte le richieste a questi file statici (percorsi, ecc.).File static non significa che l'applicazione non è dinamica. Una web app è una browser abb, successivamente il Js gestirà gli aspetti dinamici dell'app dinamica. Se non hai bisogno di generare dinamicamente i tuoi i dati, allora basta usare un Server Side Framework.

```
// define a special folder of static file

// our client
app.use(express.static("./www"));
```

```
// * means everything
app.all("*", (req, res) => {
    res.status(404).send("<h>404</h1>");
})

app.listen(8080)
```

API vs SSR

API:

- JSON
- Send data
- res.json()

SSR:

- template
- send template
- res.render()

L'architettura API è una delle più utilizzate nello sviluppo di un software di back-end. Con l'API tutti i dati verranno inviati in tipo JSON e quindi saranno compatibili con tutti gli ambienti.

Un altro metodo è quello di generare dinamicamente un intero client ad ogni richiesta, come PHP con Laravel, utilizzando il rendering di Express con un motore di rendering. L'ultimo è uno dei più complicati e non molto utilizzato per lo sviluppo del back-end al giorno d'oggi.

Route Params

A volte, il cliente vorrebbe accedere ad una specifica risorsa, o magari cancellarla o aggiornarla. Per fare ciò bisogna accedere ai route parameters. Per accedere a questi parametri bisogna usare la variabile request ed i suoi params attribute. Successivamente possiamo conoscere tutti i parametri che sono definiti nella url route. Nella url route puoi definire i nomi dei parametri con la sintassi `:paramName`

```
// not dynamic solution
app.get("/api/products/1", (req, res) => {
    res.json(products[0]);
})

// dynamic solution
app.get("api/products/:productID", (req, res) => {
    // params are all strings same of const req.params.productID
    const {productID} = req.params;
    res.json(products[Number(productID)]);
})
```

Query params

In aggiunta, il client dovrebbe inviare più informazioni riguardo la risorsa. Per fare ciò, bisogna accedere ai query parameters (o URL parameters). Per accedere a questi aprametri bisogna usare la variabile request e l'attributo query.

```
app.get("/api/v1/search", (req, res) => {
    const {search, limit} = req.query;
    let sortedProducts = [...products];
    if (search){
        sortedProducts = sortedProducts
            .filter((product)=> product.name.startsWith(search))
    }
    if (limit){
        sortedProducts = sortedProducts.slice(0, Number(limit))
    }
    if (sortedProducts.length<1){
        res.status(200).json({success:true, data:[]})
    }
    res.status(200).json({success:true, data:sortedProducts});
})
```

Middleware

Sono funzioni che esegue durante le richieste al server. Ogni middleware ha accesso agli oggetti di request e response. In express ogni cosa è un middleware ed è il cuore di express. Un middleware sta tra la request e la response. Anche un middlware ha il riferimento al middleware next ed è necessario chiamarlo. È possibile usare infiiniti middleware

```
const logger = (req, res, next) => {
    // log the request
    console.log(req.method, req.url);
    // then i can go to the next middleware
    next();
}

app.get("/", logger, (req, res) => {res.send("Home")});
app.get("/about", logger, (req, res) => {res.send("About")});

app.listen(8080)
```

Use

I middleware sono un potente strumento all'interno del quale possiamo aggiungere alcuni comportamenti basati su percorsi e metodi, come la manipolazione o il controllo dei dati in input o l'autenticazione dell'utente. Ma aggiungere queste funzioni nelle nostre singole rotte potrebbe essere un compito enorme.

Abbiamo bisogno di qualcosa che applichi un middleware a tutti i percorsi! Usare `use` permette di aggiungere un middleware basato sulla invocazione della posizione a tutte le rotte e agli altri middleware. È possibile anche definire un path!

```
app.use(logger);

// or can do also
app.use("/api", logger);
```

Post

Per impostazione predefinita non è possibile accedere al body (payload) di una richiesta POST. È necessario impostare e utilizzare un middleware di express che analizzi il corpo della richiesta in qualcosa di leggibile, ad esempio JSON. Questo è il middleware urlencoded (il vecchio bodyParser) e il middleware json.

```
app.use(express.static("./www"))

app.use(express.urlencoded({extent:false}));
app.use(express.json());

app.post("/login", (req, res) => {
    const {email} = req.body;
    if (email){
        return res.status(200).send("Welcome back"+email);
    }

    return res.status(401).send("Email must be sent)
})
```

NOTA: The flag extended : false needs to force the parsing with body-parser. This works also for PUT and DELETE methods.

Router

Router consente agli sviluppatori di raggruppare le route di richiesta in base a una logica. Inoltre, vengono utilizzati in genere nel modello MVC. L'utilizzo dei router permette di organizzare i percorsi delle richieste in gruppi e categorie! In questo modo, puoi espandere il tuo back-end più facilmente. Inoltre puoi implementare i controller e i modelli MVC (sono middleware!).

```
//router.js

const express = require("express");

const router = express.Router();
```

```
router.get("/peoples" , (req, res) => { res.send(peoples)});

router.get("/peoples/:id", (req, res) => {
res.send(peoples[req.params.id])});
module.exports = router;

//app.js

const express = require("express")
const app = express()
const mainRouter = require("./router");

app.use(express.static("./www"));

//parse form data
app.use(express.urlencoded({extent:false}));

app.use(express.json());

//load router

app.use("/api", mainRouter);
```

WS & Socket.io

Quando un utente naviga sul Web, il browser (client) invia determinate richieste tramite percorsi REST ("GET", "POST", ecc.) al server che ospita il sito Web a cui stanno tentando di accedere. Il server riceve le richieste e invia le informazioni come risposte al client, che riceve e restituisce le informazioni sulla risposta sulla pagina. Se hai bisogno di aggiornare alcune informazioni, devi fare più richieste al server in un intervallo di tempo. Questo approccio è chiamato polling. Le WebSocket sono diverse perché funzionano tenendo sempre aperta la connessione dal client al server. In questo modo, il server può inviare informazioni al client, anche in assenza di una richiesta esplicita da parte del cliente. I client possono comunque effettuare richieste HTTP al server come di consueto; tuttavia, la connessione WebSocket consente una comunicazione costante nel caso in cui vengano creati nuovi dati e quindi debbano essere renderizzati lato client.

Una connessione inizia con un handshake HTTP che richiede un aggiornamento per il protocollo, questo è chiamato WebSocket Handshake request. L'handshake inizia con una richiesta/risposta HTTP, consentendo ai server di gestire le connessioni HTTP e le connessioni WebSocket sulla stessa porta. Una volta stabilita la connessione, la comunicazione passa a un protocollo binario bidirezionale che non è conforme al protocollo HTTP. Oltre alle intestazioni di aggiornamento, il client invia un'intestazione Sec-WebSocket-Key contenente byte casuali con codifica base64 e il server risponde con un hash della chiave nell'intestazione Sec-WebSocket-Accept. Ciò ha lo scopo di impedire a un proxy di memorizzazione nella cache di inviare nuovamente una conversazione WebSocket precedente e non fornisce alcuna autenticazione, privacy o integrità.

Una volta stabilita la connessione, il client e il server possono inviare dati WebSocket o frame di testo avanti e indietro in modalità full-duplex. I dati sono minimamente inquadrati, con una piccola intestazione seguita

dal carico utile. Le trasmissioni WebSocket sono descritte come "messaggi", in cui un singolo messaggio può essere opzionalmente suddiviso su più frame di dati

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
    Origin: http://example.com


HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
    Sec-WebSocket-Protocol: chat
```

I WebSocket funzionano a livello di applicazione e possono funzionare senza un server specifico. In effetti, un server WebSocket può essere eseguito utilizzando un server HTTP esistente o autonomo. Per definire una connessione WebSocket devi definire un URL usando il protocollo ws://, invece di http://. Come HTTP, i WebSocket supportano la crittografia SSL e puoi specificarla utilizzando il protocollo wss://.

L'associazione immediata di alcuni gestori di eventi alla connessione consente di sapere quando la connessione è stata aperta, ha ricevuto messaggi in arrivo o si è verificato un errore.

```
var connection = new WebSocket('ws://html5rocks.websocket.org/echo')
// When the connection is open, send some data to the server
connection.onopen = function () {
    // Send the message ping to the server
    connection.send('Ping');
}

// log errors
connection.onerror = function (error) {
    console.log('WebSocket Error' + error);
}

// log messages from the server
connection.onmessage = function (e) {
    console.log('Server: ' + e.data);
}
```

`ws` è una libreria molto utilizzata per usare WebSockets in applicazioni Node.js per integrare WebSockets nelle applicazioni (da installare con npm)

```
// stand alone server

const ws = require('ws')
const connection = new ws.Server({port: 8080})

connection.on('connection', ws => {
    connection.on('message', message => {
        console.log(`Receuved message => ${message}`)
    });
    connection.send('Message from server');
})

// express server
const ws = require('ws')
const express = require('express');

const app = express();

// setup a headless websocket server
const wsServer = new ws.Server({noServer: true});
wsServer.on('connection', socket => {
    socket.on('message', message => console.log(message));
})

// 'server' is a vanilla Node.js HTTP server

server.on('upgrade', (req, socket, head) => {
    wsServer.handleUpgrade(req, socket, head, socket => {
        wsServer.emit('connection', socket, request);
    })
})
```

Socket.io

È un protocollo nuovo ma il suo utilzzo nelle IoT app sta crescendo. Tuttavia vi sono dei problemi con le ws:

- Non pensato per la comunicazione da server a server, forse è preferibile un socket UDP di basso livello
- Non pensato per la comunicazione da client a client (p2p), per questo c'è WebRTC!
- Se si utilizzano proxy, questi devono essere compatibili con WebSocket
- Il server deve supportare i WebSocket e il relativo handshake La lingua di backend deve supportare i WebSocket
- IL BROWSER DEVE supportare i WebSocket (non tutti i browser supportano ws)

Quindi, abbiamo bisogno di qualcosa che consenta lo sviluppo di una comunicazione full-duplex in tempo reale senza questi problemi!

Socket.IO è una libreria che consente la comunicazione a bassa latenza, bidirezionale e basata su eventi tra un client e un server. Si basa sul protocollo WebSocket e fornisce garanzie aggiuntive come il fallback al

long polling HTTP o la riconnessione automatica.

Il canale bidirezionale tra il server Socket.IO (Node.js) e il client Socket.IO (browser, Node.js o un altro linguaggio di programmazione) viene stabilito con una connessione WebSocket quando possibile e utilizzerà il polling HTTP lungo come fallback .

La base di codice di Socket.IO è divisa in due livelli distinti:

▸ l'architettura di basso livello: quello che chiamiamo Engine.IO, il motore dentro Socket.IO ▸ l'API di alto livello: Socket.IO stesso

Engine.IO è responsabile di stabilire la connessione di basso livello tra il server e il client. Gestisce:

▸ I vari trasporti (WebSocket o HTTP long polling) ▸ Il meccanismo di aggiornamento ▸ Il rilevamento della disconnessione

Per impostazione predefinita, il client stabilisce la connessione con il trasporto di polling lungo HTTP. Sebbene WebSocket sia chiaramente il modo migliore per stabilire una comunicazione bidirezionale, l'esperienza ha dimostrato che non è sempre possibile stabilire una connessione WebSocket, a causa di proxy aziendali, firewall personale, ecc.

Per riassumere, Engine.IO si concentra sull'affidabilità e sull'aggiornamento, il cliente dovrà:

▸ assicurarsi che il buffer in uscita sia vuoto ▸ mettere il trasporto corrente in modalità di sola lettura ▸ provare a stabilire una connessione con l'altro trasporto ▸ in caso di successo, chiudere il primo trasporto

La connessione Engine.IO è considerata chiusa quando:

▸ una richiesta HTTP (GET o POST) non riesce (ad esempio, quando il server viene spento) ▸ la connessione WebSocket viene chiusa (ad esempio, quando l'utente chiude la scheda nel proprio browser) ▸ socket.disconnect() viene chiamato sul lato server o sul lato client

C'è anche un meccanismo di heartbeat che controlla che la connessione tra il server e il client sia ancora attiva e funzionante.

Utilizzo

```
// stand alone server

const { Server } = require("socket.io");
const io = new Server({ /* options */ })

io.on("connection", (socket) => {
    // ...
});

io.listen(3000);

// with http server
const { createServer } = require("http");
const httpServer = createServer();
const io = new Server(httpServer, { /*options*/});
```

```
io.on("connection", (socket) => {
    // ....
})
```

```
// with express
const express = require("express)
const { createServer } = require("http");
const { Server } = require("socket.io");
const app = express();
const httpServer = createServer(app);
const io = new Server(httpServer, { /*options*/});

io.on("connection", (socket) => {
    // ....
})

httpServer.listen(8080);
```

Una Socket è la classe fondamentale per interagire con il client. Eredita tutti i metodi di Node.js EventEmitter, come emit, on, once o removeListener.

```
// ogni nuova connessione è assegnata a 20 caratteri random

// server-side
io.on("connection", (socket) => {
    console.log(socket.id)
})

// client side
socket.on("connect", () => {
    console.log(socket.id)
});

// sulla creazione, la socket joina la stanza identificata dal proprio id
che significa che puoi usare per messaggi privati

io.on("connection", socket => {
    socket.on("private message", (anotherSocketId, msg) => {
        socket.to(anotherSocketId).emit("private message", socket.id,
msg);
    })
})
```

L'idea è quella di emettere ed elencare alcuni eventi che lo sviluppatore può definire. Ad esempio, nel client e nel server si potrebbe definire un evento chiamato “onMessage” e usarlo per scambiare messaggi, emettendo questo evento con un payload!

Broadcasting

Significa inviare messaggi a tutti i client connessi. Può essere fatto a livelli multpli, possiamo inviare i messaggi a tutti i client connessi, ai client sul namespace e ad una room in particolare. Per effettuare un broadcast a tutti gli eventi bisogna usare il metodo `io.socket.emit`.

```
io.on('connection',(socket) => {
    clients++;
    io.sockets.emit('broadcast', {description: clients + 'clients
connected!' });
    socket.on('disconnect', () => {
        clients--;
        io.socket.emit('broadcast', { description: clients + 'clients
connected!' });
    });
});
```

Se vogliamo inviare un evento a tutti ma il client lo ha causato, possiamo usare `socket.broadcast.emit`

```
io.on('connection', (socket) => {
    clients++;
    socket.emit('newclientconnect', {description: 'Hey, welcome!' });
    socket.broadcast.emit('newclientconnect', {description: client +
'clients connected' })
    socket.on ('disconnect', () => {
        clients--;
        socket.broadcast.emit('newclientconnect', { description: clients +
'clients connected!' })
    })
})
```

Namespaces

Socket.IO ti consente di assegnare "namespace" i tuoi socket, il che significa essenzialmente assegnare diversi endpoint o percorsi. Questa è una funzione utile per ridurre al minimo il numero di risorse (connessioni TCP) e allo stesso tempo separare le preoccupazioni all'interno dell'applicazione introducendo la separazione tra i canali di comunicazione. Più spazi dei nomi condividono effettivamente la stessa connessione WebSocket, salvandoci così le porte socket sul server.

Lo spazio dei nomi radice '/' è lo spazio dei nomi predefinito, a cui si uniscono i client se uno spazio dei nomi non è specificato dal client durante la connessione al server. Tutte le connessioni al server che utilizzano il lato client dell'oggetto socket vengono effettuate nello spazio dei nomi predefinito. Di conseguenza per

connettere i client al namespace, si ha la necessità di dare un namespace come argomento al costruttore io chiamato per creare una connessione ed un socket object sul lato client.

```
const nsp = io.of('/my-namespace');
nsp.on('connection', function(socket){
    console.log('someone connected');
    nsp.emit('hi', 'Hello everyone');
})
```

Rooms

All'interno di ogni spazio dei nomi, puoi anche definire canali arbitrari a cui i socket possono unirsi e abbandonare. Questi canali sono chiamati stanze. Le stanze sono usate per separare ulteriormente i concetti. Le stanze condividono anche la stessa connessione socket come gli spazi dei nomi. Una cosa da tenere a mente durante l'utilizzo delle stanze è che possono essere unite solo sul lato server.

È possibile chiamare il metodo join sulla presa per sottoscrivere la presa a un determinato canale/stanza. Ad esempio, creiamo stanze chiamate 'room-' e uniamoci ad alcuni clienti. Non appena questa stanza è piena, crea un'altra stanza e unisciti ai clienti lì.

Puoi anche implementarlo in spazi dei nomi personalizzati allo stesso modo. Per lasciare una stanza, devi chiamare la funzione leave proprio come hai chiamato la funzione join sul socket.