

Introduzione a JavaScript

JavaScript è un linguaggio di programmazione creato nel 1995 da **Brendan Eich** della **Netscape**. Originariamente chiamato **Livescript** o **Mocha**, è stato sviluppato come linguaggio semplice per i non sviluppatori. Nonostante il nome, **JavaScript** non è correlato a **Java**; il nome fu scelto solo per motivi di marketing. Nel 1996, divenne uno standard sotto l'**ECMA** (European Computer Manufacturer's Association) e il linguaggio è conosciuto anche come **ECMAScript**. L'ultima versione dello standard è **ECMAScript 2023**.

Ruolo di JavaScript nel Web

JavaScript è fondamentale nella **programmazione web**. Si usa per rendere le pagine web interattive e dinamiche. Le principali aree d'uso sono:

- **Struttura** (HTML)
- **Presentazione** (CSS)
- **Comportamento** (JS)

JavaScript può operare sia sul **client-side** che sul **server-side** (con **NodeJs**). È utilizzato per molte funzionalità web moderne come le ricerche istantanee, le chat in tempo reale, e le comunicazioni asincrone tramite **AJAX** (Asynchronous JavaScript and XML).

Caratteristiche del Linguaggio

- **Dynamic**: JavaScript non è compilato, ma eseguito in una "macchina virtuale".
- **Loosely typed**: Le variabili non richiedono una dichiarazione del tipo.
- **Case-sensitive**: Fa distinzione tra maiuscole e minuscole.

JavaScript include anche un **garbage collector** per la gestione automatica della memoria, eliminando le variabili non più necessarie.

Ambito (Scope)

- **var**: Ha uno **scope di funzione** o **global scope**. Se dichiarata all'interno di una funzione, la variabile è visibile in tutta la funzione, indipendentemente da blocchi o istruzioni condizionali all'interno della funzione. Se dichiarata al di fuori di una funzione, è globale.
- **let**: Ha uno **scope di blocco**. È visibile solo all'interno del blocco `{ }` in cui è stata dichiarata (come in un ciclo `for`, un blocco `if`, ecc.), anche se dichiarata all'interno di una funzione.

Array

- Creare un array (metodi equivalenti)
 - `let arr = [element0, element1, ..., elementN];`
- Modificare un membro
 - `myFirstArray[0] = "nuovo valore";`
- Aggiungere un membro
 - `myFirstArray.push("ciao")` //aggiunge alla fine
 - `myFirstArray.unshift("ciao")` //aggiunge all'inizio
 - `myFirstArray[10] = "ciao";` // aggiunge al decimo posto (length sarà almeno 11)
- Rimuovere un membro
 - `myFirstArray.pop()` // rimuove ultimo elemento ritornandolo
 - `myFirstArray.shift()` // rimuove il primo elemento ritornandolo
 - `delete myFirstArray[10]` // rimuove l'elemento ma non sposta gli indici dell'array
- Lunghezza dell'array
 - `myFirstArray.length`
- Svuotare un array
 - `myFirstArray = []`
 - `myFirstArray.length = 0`

Programmazione WEB

Stringhe

```
let s = "Ciao a tutti"; // creiamo una stringa
```

```
s = "ciao a \n tutti"; // se è lunga possiamo andare a capo
```

```
s.indexOf(" a "); // ritorna 4 (prima occorrenza)  
s.slice(1); // ritorna "iao a tutti" (non modifica la stringa)  
s.trim(); // leva whitespaces ad inizio e fine stringa  
s.charAt(1); // ritorna "i" (carattere a indice 1)  
s.toUpperCase(); // ritorna la stringa in maiuscolo  
s.toLowerCase(); // ritorna la stringa in minuscolo
```

Programmazione WEB

Metodi di Iterazione per gli Array in JavaScript

In JavaScript, esistono diversi metodi per iterare e manipolare gli array. Ecco una panoramica dei più comuni, inclusi **reduce** e **map**.

1. forEach

Esegue una funzione per ciascun elemento dell'array.

```
let array = [1, 2, 3];
array.forEach(function(element) {
  console.log(element);
});
// Output: 1 2 3
```

2. map

Crea un nuovo array con i risultati della funzione applicata a ogni elemento dell'array chiamante.

```
let array = [1, 2, 3];
let mappedArray = array.map(function(element) {
  return element * 2;
});
console.log(mappedArray); // Output: [2, 4, 6]
```

Map (array method)

- Metodo che serve a "convertire" (mappare) un array in un altro
 - Item: i-simo oggetto dell'array
 - Index: indice dell'oggetto (partendo da 0)
 - Array: l'array
- Esempio: dato un array di stringhe, generiamo un array contenente quanto sono lunghe

```
let a = ["pippo", "pluto", "paperino"];
```

```
a.map( (item, index, array) => item.length);  
// ritorna [5, 5, 8]
```

Programmazione WEB

3. filter

Crea un nuovo array con tutti gli elementi che superano il test implementato dalla funzione fornita.

```
let array = [1, 2, 3, 4];  
let filteredArray = array.filter(function(element) {  
    return element > 2;  
});  
console.log(filteredArray); // Output: [3, 4]
```

4. reduce

Applica una funzione su un accumulatore e su ogni elemento dell'array (da sinistra a destra) per ridurlo a un singolo valore.

```
let array = [1, 2, 3, 4];
let sum = array.reduce(function(accumulator, currentValue) {
    return accumulator + currentValue;
}, 0);
console.log(sum); // Output: 10
```

5. reduceRight

Funziona come `reduce`, ma applica la funzione da destra a sinistra.

```
let array = [1, 2, 3, 4];
let sum = array.reduceRight(function(accumulator, currentValue) {
    return accumulator + currentValue;
}, 0);
console.log(sum); // Output: 10
```

6. some

Verifica se almeno un elemento dell'array soddisfa il test implementato dalla funzione fornita. Ritorna un booleano.

```
let array = [1, 2, 3];
let hasEvenNumber = array.some(function(element) {
    return element % 2 === 0;
});
console.log(hasEvenNumber); // Output: true
```

7. every

Verifica se tutti gli elementi dell'array soddisfano il test implementato dalla funzione fornita. Ritorna un booleano.

```
let array = [1, 2, 3];
let allEvenNumbers = array.every(function(element) {
    return element % 2 === 0;
});
console.log(allEvenNumbers); // Output: false
```

8. find

Ritorna il primo elemento dell'array che soddisfa il test implementato dalla funzione fornita. Altrimenti, ritorna `undefined`.

```
let array = [1, 2, 3, 4];
let foundElement = array.find(function(element) {
    return element > 2;
});
console.log(foundElement); // Output: 3
```

9. findIndex

Ritorna l'indice del primo elemento dell'array che soddisfa il test implementato dalla funzione fornita. Altrimenti, ritorna `-1`.

```
let array = [1, 2, 3, 4];
let foundIndex = array.findIndex(function(element) {
    return element > 2;
});
console.log(foundIndex); // Output: 2
```

10. indexOf

Ritorna il primo indice in cui è presente un determinato elemento, oppure `-1` se l'elemento non è presente nell'array.

```
let array = [1, 2, 3, 4];
let index = array.indexOf(3);
console.log(index); // Output: 2
```

11. includes

Verifica se un array contiene un determinato elemento, restituendo `true` o `false`.

```
let array = [1, 2, 3, 4];
let containsElement = array.includes(3);
console.log(containsElement); // Output: true
```

12. sort

Ordina gli elementi di un array e ritorna l'array ordinato.

```
let array = [4, 2, 3, 1];
array.sort();
console.log(array); // Output: [1, 2, 3, 4]
```

13. slice

Ritorna una copia superficiale di una porzione di un array in un nuovo array, selezionata dall'inizio a una fine (non inclusa) indicati dagli indici.

```
let array = [1, 2, 3, 4];
let slicedArray = array.slice(1, 3);
console.log(slicedArray); // Output: [2, 3]
```

Slicing



- `slice(start_index, end_index)` ritorna una porzione dell'array (non modifica l'array)

```
let myArray = ["a", "b", "c", "d", "e"];
myArray = myArray.slice(1, 4); // ritorna [ "b", "c", "d"]
```

- `splice(index, n_elementi)` rimuove "n_elementi" partendo da "index", ritornandoli (modifica l'array).

```
let myArray = ["1", "2", "3", "4", "5"];
myArray.splice(3, 2); //ritorna ["4","5"]
// ora myArray e' ["1", "2", "3",]
```

14. splice

Cambia il contenuto di un array rimuovendo o sostituendo elementi esistenti e/o aggiungendo nuovi elementi in loco.

```
let array = [1, 2, 3, 4];  
array.splice(1, 2, 'a', 'b');  
console.log(array); // Output: [1, 'a', 'b', 4]
```

Eventi



onblur/onfocus	Un elemento prende/perde il focus
onchange	Il contenuto di un form cambia
onclick	Mouse click
onerror	Quando c'è un errore nel caricamento di immagini o del documento
onload	La pagina ha finito di caricarsi
onkeydown/onkeypress/onkeyup	Un tasto viene premuto/tenuto premuto/rilasciato
onmousedown/onmouseup	un bottone del mouse è premuto/rilasciato
onmousemove/onmouseout/onmouseover	Il mouse si è spostato/spostato fuori da un elemento/spostato sopra un elemento
onsubmit	E' stato premuto il pulsante submit di un form

Programmazione WEB

Programmazione Web: Interazioni del Browser, Richieste HTTP e Tipi di Server

Interazioni del Browser

1. **Rendering delle Pagine Web:** Quando un utente inserisce un URL nella barra degli indirizzi del browser o clicca su un link, il browser invia una richiesta HTTP al server. Il server restituisce i dati (spesso HTML, CSS e JavaScript), che il browser interpreta e visualizza come una pagina web. Questo processo include:
 - **Parsing:** Il browser analizza il codice HTML e CSS.
 - **Rendering:** Il browser crea il DOM (Document Object Model) e il CSSOM (CSS Object Model) e li combina per creare il Render Tree.

- **Layout:** Il browser calcola la posizione e la dimensione di ogni elemento nella pagina.
- **Painting:** Il browser disegna i pixel sullo schermo.
- 2. **Esecuzione di JavaScript:** I browser hanno motori JavaScript integrati (come V8 per Chrome e SpiderMonkey per Firefox) che eseguono script. JavaScript può manipolare il DOM e il CSSOM, aggiornando dinamicamente la pagina web.
- 3. **Gestione delle Risorse:** Il browser gestisce risorse come immagini, fogli di stile e script. Utilizza la cache per migliorare le prestazioni, memorizzando risorse che sono state precedentemente scaricate.

Richieste HTTP

1. **Metodi HTTP:** Le richieste HTTP possono utilizzare vari metodi, i più comuni sono:
 - **GET:** Richiede dati dal server.
 - **POST:** Invia dati al server.
 - **PUT:** Aggiorna i dati sul server.
 - **DELETE:** Elimina dati sul server.
2. **Struttura della Richiesta HTTP:**
 - **Request Line:** Contiene il metodo, l'URL e la versione del protocollo (es. **GET /index.html HTTP/1.1**).
 - **Header:** Contiene metadati come tipo di contenuto e informazioni di autenticazione.
 - **Body:** Contiene i dati inviati con la richiesta (solo per POST e PUT).
3. **Struttura della Risposta HTTP:**
 - **Status Line:** Contiene il codice di stato e la versione del protocollo (es. **HTTP/1.1 200 OK**).
 - **Header:** Contiene metadati sulla risposta.
 - **Body:** Contiene i dati richiesti, come HTML, JSON, ecc.

Tipi di Server: Apache e Nginx

1. **Apache:**
 - **Architettura:** Utilizza un modello di processo o thread, creando un nuovo processo per ogni richiesta o un thread per ogni connessione.
 - **Configurazione:** Configurabile tramite file **.htaccess** e **httpd.conf**. Supporta moduli per estendere la funzionalità.
 - **Uso:** Comunemente usato per applicazioni web dinamiche e per la sua robustezza e configurabilità.
2. **Nginx:**
 - **Architettura:** Utilizza un modello di evento asincrono e non bloccante, gestendo molte connessioni simultaneamente con un uso efficiente della memoria.
 - **Configurazione:** Configurabile tramite file **nginx.conf**. Ideale per la gestione di proxy e bilanciamento del carico.
 - **Uso:** Spesso utilizzato come server proxy inverso o per servire contenuti statici ad alte prestazioni.

Markup: Tag HTML, Standard e Uso nello Sviluppo Web

1. Tag Comuni:

- **<html>**: Racchiude l'intero documento HTML.
- **<head>**: Contiene metadati come il titolo e i fogli di stile.
- **<body>**: Contiene il contenuto visibile della pagina.
- **<h1>** - **<h6>**: Intestazioni di diverse dimensioni.
- **<a>**: Crea collegamenti ipertestuali.
- ****: Inserisce immagini.
- **<form>**: Crea moduli per l'invio di dati.

Standard HTML

1. HTML5:

- **Elementi Semantici**: `<header>`, `<footer>`, `<article>`, `<section>`, che migliorano la struttura e l'accessibilità della pagina.
- **Form di Nuova Generazione**: Nuovi attributi e tipi di input come `placeholder`, `required`, `email`, `date`.
- **API**: Nuove API come `localStorage`, `sessionStorage`, `Canvas`, e `Geolocation`.

2. Compatibilità:

- I browser moderni supportano HTML5, ma è importante testare la compatibilità con browser più vecchi se necessario.

Funzionalità del Browser: Individuazione, Richiesta, Verifica e Visualizzazione

1. Individuazione:

- **DNS Lookup**: Il browser utilizza DNS per risolvere un URL in un indirizzo IP.

2. Richiesta:

- **Costruzione della Richiesta**: Il browser crea una richiesta HTTP e la invia al server.

3. Verifica:


- **Verifica SSL/TLS**: Se l'URL utilizza HTTPS, il browser verifica il certificato SSL/TLS per garantire una connessione sicura.

4. Visualizzazione:

- **Rendering**: Il browser elabora HTML, CSS e JavaScript per visualizzare la pagina.
- **Interazione**: JavaScript gestisce interazioni dinamiche come animazioni e manipolazioni del DOM.

Stile con CSS

```
<!DOCTYPE html>
<html lang="it">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Menu di Navigazione</title>
  <style>
    .menu {
      list-style-type: none; /* Rimuove i pallini predefiniti */
      padding: 0;
      margin: 0;
      background-color: #333;
    }
    .menu li {
      display: inline;
      margin-right: 20px;
    }
    .menu a {
      color: white;
      text-decoration: none;
      padding: 10px;
      display: inline-block;
    }
    .menu a:hover {
      background-color: #575757;
    }
  </style>
</head>
<body>
  <ul class="menu">
    <li><a href="#home">Home</a></li>
    <li><a href="#about">About</a></li>
    <li><a href="#services">Services</a></li>
    <li><a href="#contact">Contact</a></li>
  </ul>
</body>
</html>
```



Come Calcolare la Specificità

La specificità viene calcolata utilizzando una serie di regole che considerano il tipo di selettore utilizzato. Ogni tipo di selettore ha un valore specifico associato che contribuisce al punteggio di specificità totale. Ecco una panoramica di come calcolare la specificità:

1. **Selettori di ID:** Ogni selettore ID ha il valore di 1 nella posizione dei selettori di ID.
2. **Selettori di Classe, Attributo e Pseudo-classe:** Ogni selettore di classe, attributo o pseudo-classe ha il valore di 1 nella posizione dei selettori di classe.
3. **Selettori di Tipo (Elementi) e Pseudo-elementi:** Ogni selettore di tipo o pseudo-elemento ha il valore di 1 nella posizione dei selettori di tipo.

La specificità è espressa come un valore a quattro parti: `[a, b, c, d]`, dove:

- `a` = Numero di selettori ID
- `b` = Numero di selettori di classe, attributo e pseudo-classe
- `c` = Numero di selettori di tipo (elementi) e pseudo-elementi
- `d` = Numero di selettori inline (stili inline)

Esempi di Calcolo della Specificità

1. Selettore di Tipo (Elementi)

Esempio: `div`

- Specificità: `[0, 0, 1, 0]`

2. Selettore di Classe

Esempio: `.class-name`

- Specificità: `[0, 1, 0, 0]`

3. Selettore di ID

Esempio: `#id-name`

- Specificità: `[0, 0, 0, 1]`

4. Selettore di Attributo

Esempio: `[type="text"]`

- Specificità: `[0, 1, 0, 0]`

5. Selettore di Attributo e Classi

Esempio: `input[type="text"].class-name`

- Specificità: `[0, 2, 0, 0]` (1 per `[type="text"]` e 1 per `.class-name`)

6. Selettore Combinato

Esempio: `div.class-name #id-name`

- Specificità: `[0, 1, 1, 1]` (1 per `.class-name`, 1 per `div`, e 1 per `#id-name`)

7. Stile Inline

Esempio: `<div style="color: red;">`

- Specificità: `[0, 0, 0, 1]` (Gli stili inline hanno la specificità più alta e sovrascrivono gli altri selettori).

Struttura ad Albero del DOM

La **struttura ad albero del DOM** rappresenta il documento HTML come una gerarchia di nodi, dove ogni nodo è un elemento o un oggetto nel documento. Questo albero riflette le relazioni genitore-figlio tra gli elementi.

Esempio di Struttura ad Albero del DOM:

html

Copia codice

```
<!DOCTYPE html>
<html>
  <head>
    <title>Pagina di Esempio</title>
  </head>
  <body>
    <div class="container">
      <h1 id="main-title">Benvenuto</h1>
      <p class="description">Questo è un esempio di struttura del
DOM.</p>
    </div>
  </body>
</html>
```

Struttura ad Albero:

- `html` (radice)
 - `head`
 - `title`
 - `body`
 - `div.container`
 - `h1#main-title`
 - `p.description`

Stile Inline

Gli **stili inline** sono quelli applicati direttamente all'elemento HTML tramite l'attributo **style**. Questi stili hanno una specificità molto alta rispetto ai selettori esterni.

Esempio di Stile Inline:

html

Copia codice

```
<p style="color: blue; font-size: 20px;">Testo con stile inline.</p>
```

Specificità di Stile Inline:

- Specificità: `[0, 0, 0, 1]` (Gli stili inline sovrascrivono gli stili definiti nei fogli di stile esterni o interni).

Unità di Misura per la Dimensione del Testo

Le unità di misura per la dimensione del testo in CSS possono essere suddivise in due categorie principali: assolute e relative.

Unità Assolute

- **px (pixel)**: Misura fissa e non scalabile. Ad esempio, `font-size: 16px;`.
- **pt (punto)**: Unità di misura tipografica, 1 pt è pari a 1/72 di pollice. Ad esempio, `font-size: 12pt;`.
- **cm (centimetro)**: Misura fisica del testo. Ad esempio, `font-size: 1cm;`.
- **mm (millimetro)**: Misura fisica più piccola. Ad esempio, `font-size: 5mm;`.
- **in (pollice)**: Misura fisica del testo. Ad esempio, `font-size: 1in;`.

Unità Relative

- **em**: Relativa alla dimensione del font del genitore. Ad esempio, se il font-size del genitore è 16px, `font-size: 1.5em;` sarà 24px.
- **rem**: Relativa alla dimensione del font radice (`<html>`). Ad esempio, se `font-size` del radice è 16px, `font-size: 2rem;` sarà 32px.
- **% (percentuale)**: Relativa alla dimensione del font del genitore. Ad esempio, `font-size: 120%;` aumenterà la dimensione del font del 20% rispetto al font-size del genitore.

Colori e Famiglie di Caratteri

Colori

I colori possono essere definiti in CSS usando vari formati:

- **Nomi di Colore**: Ad esempio, `color: red;`
- **Codice Esadecimale**: Ad esempio, `color: #ff0000;`

- **RGB:** Ad esempio, `color: rgb(255, 0, 0);`
- **RGBA:** Ad esempio, `color: rgba(255, 0, 0, 0.5);` (con opacità)
- **HSL:** Ad esempio, `color: hsl(0, 100%, 50%);`
- **HSLA:** Ad esempio, `color: hsla(0, 100%, 50%, 0.5);` (con opacità)

Dettagli sui Selettori

- **:link:** Applica gli stili ai collegamenti non ancora visitati. Questo selettore viene applicato per primo e solo se il link non è stato cliccato.
- **:visited:** Applica gli stili ai collegamenti che l'utente ha già visitato. Questo selettore viene applicato dopo **:link**.
- **:hover:** Applica gli stili quando l'utente passa il cursore sopra il link. Questo selettore ha la priorità rispetto a **:link** e **:visited**.
- **:focus:** Applica gli stili quando il link riceve il focus (ad esempio, navigando con la tastiera). Questo selettore è utile per l'accessibilità.
- **:active:** Applica gli stili quando il link è attivamente cliccato. Questo selettore è applicato solo durante il momento in cui l'utente sta cliccando sul link.

Ordine di Applicazione


Gli stili dei selettori per link vengono applicati in questo ordine:

1. **:link**
2. **:visited**
3. **:hover**
4. **:focus**
5. **:active**

Questo significa che se un link è sia visitato che si trova sotto il cursore, verrà applicato lo stile di **:hover**, ma se è cliccato, verrà applicato lo stile di **:active**.

Questi selettori e tecniche ti permettono di creare collegamenti visivamente distintivi e interattivi, migliorando l'esperienza utente sul tuo sito web. Se hai altre domande o hai bisogno di ulteriori chiarimenti, fammelo sapere!

```
<!DOCTYPE html>
<html lang="it">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Stili per Link</title>
  <style>
    a:link {
      color: blue; /* Colore dei collegamenti non visitati */
      text-decoration: none;
    }
    a:visited {
      color: purple; /* Colore dei collegamenti visitati */
    }
    a:hover {
      color: red; /* Colore dei collegamenti al passaggio del mouse */
      text-decoration: underline;
    }
    a:focus {
      outline: 2px solid orange; /* Contorno per il focus (tastiera) */
    }
    a:active {
      color: green; /* Colore dei collegamenti mentre vengono cliccati */
    }
  </style>
</head>
<body>
  <p>
    <a href="https://www.example.com">Visita il nostro sito</a>
  </p>
</body>
</html>
```



Layout Fisso vs. Layout Fluido

Layout Fisso

Un layout fisso utilizza dimensioni fisse per gli elementi, che non si adattano alle dimensioni dello schermo o alla finestra del browser. Questo tipo di layout ha una larghezza definita in pixel (px) e non cambia in base alle dimensioni dello schermo.

Svantaggi:

1. **Responsività Limitata:** Non si adatta bene a schermi di dimensioni diverse, il che può portare a problemi di visualizzazione su dispositivi mobili o schermi grandi.
2. **Scroll Orizzontale:** Potrebbe essere necessario scorrere orizzontalmente su schermi più piccoli.
3. **Mantenimento:** Richiede ulteriori aggiustamenti e test per garantire che il design funzioni correttamente su diversi dispositivi.

Layout Fluido

Un layout fluido utilizza percentuali anziché valori fissi per definire la larghezza degli elementi. Gli elementi si adattano in base alla larghezza della finestra del browser.

Vantaggi:

1. **Adattabilità:** Gli elementi si adattano automaticamente alla larghezza della finestra del browser, migliorando l'esperienza su dispositivi mobili e schermi di diverse dimensioni.
2. **Design Reattivo:** Migliora la leggibilità e l'usabilità su vari dispositivi.
3. **Flessibilità:** Più facile da mantenere e aggiornare per diversi formati di schermo.

Svantaggi:

1. **Complessità:** Può essere difficile mantenere il controllo preciso sui posizionamenti degli elementi.
2. **Design Inconsistente:** Potrebbe non apparire come previsto su schermi molto grandi o molto piccoli.
3. **Test:** Richiede test su diversi dispositivi e risoluzioni per assicurare una buona esperienza utente.

Media Query

Le media query sono una funzionalità di CSS che consente di applicare stili differenti in base alle caratteristiche del dispositivo, come la larghezza dello schermo, l'orientamento, e la risoluzione.

Caratteristiche:

1. **Condizioni Basate su Dimensioni:** Cambia gli stili a seconda della larghezza o altezza della finestra del browser.
2. **Supporto per Dispositivi Differenti:** Adatta il layout per schermi piccoli (smartphone) e grandi (desktop).
3. **Orientamento:** Gestisce anche orientamenti di schermo orizzontali e verticali.

```
/* Media Query per schermi piccoli */
@media (max-width: 768px) {
  .main-content {
    flex-direction: column;
  }
  .sidebar, .article {
    width: 100%;
    margin-bottom: 20px;
  }
}
```

Breakpoint e Media Query

Cos'è un Breakpoint?

Un breakpoint è una larghezza specifica del viewport (la finestra di visualizzazione) alla quale il layout di una pagina web cambia. Questi punti di interruzione permettono di adattare il design e la disposizione degli elementi in base alla dimensione dello schermo del dispositivo.

Definizione dei Breakpoint

I breakpoint sono definiti all'interno delle media query utilizzando valori numerici che rappresentano la larghezza del viewport. Quando il viewport raggiunge o supera questi valori, vengono applicati stili specifici definiti nella media query.

Viewport e Dispositivi Mobili

Il **viewport** è la parte visibile della pagina web su un dispositivo. I dispositivi mobili spesso utilizzano valori di viewport fissi per garantire che i contenuti si adattino correttamente a schermi di dimensioni diverse. Tuttavia, il viewport fisico può variare notevolmente rispetto alla dimensione dichiarata del dispositivo.

Dichiarazione del Viewport nel Meta Tag:

html

Copia codice

```
<meta name="viewport" content="width=device-width,
initial-scale=1.0">
```

- **width=device-width**: Imposta la larghezza del viewport per essere uguale alla larghezza del dispositivo.

- **initial-scale=1.0**: Imposta il livello di zoom iniziale a 1.

Questa dichiarazione aiuta a garantire che il layout della pagina sia visualizzato correttamente sui dispositivi mobili e non venga scalato o ridimensionato in modo inappropriato.

Posizionamento Item con CSS Grid

Il **CSS Grid Layout** è un potente sistema di layout che permette di posizionare gli elementi all'interno di una griglia bidimensionale. Con CSS Grid, puoi definire righe e colonne nella tua pagina e posizionare gli elementi in modo preciso all'interno di questa griglia.

Sistema a Griglia (Grid System)

Il **grid system** è un metodo comune per strutturare layout web in modo che il contenuto possa adattarsi a diverse dimensioni dello schermo. In web development, le griglie aiutano a posizionare gli elementi in righe e colonne, creando un layout organizzato e coerente.

Caratteristiche del Sistema a Griglia:

- **Definizione di righe e colonne**: Le griglie possono essere create utilizzando CSS Grid o framework come Bootstrap.
- **Spaziatura e allineamento**: Le griglie gestiscono spazi e allineamenti tra elementi.
- **Responsive Design**: Le griglie possono essere adattate a diverse risoluzioni e dimensioni dello schermo.

var() in CSS

La funzione **var()** è utilizzata per applicare variabili CSS definite precedentemente. Permette di accedere ai valori delle custom properties nel tuo CSS, migliorando la modularità e la riusabilità del codice.

Bootstrap

Bootstrap è un framework CSS open-source che semplifica la creazione di siti web responsivi e mobili. Offre una serie di strumenti e componenti predefiniti per costruire layout e interfacce utente.

Caratteristiche Principali di Bootstrap:

- **Sistema a Griglia**: Permette di creare layout responsivi con righe e colonne.
- **Design Responsivo**: Adatta automaticamente il design per diverse dimensioni di schermo.
- **Mobile-First**: Approccio che prioritizza la progettazione per dispositivi mobili prima di passare ai desktop.
- **Compatibilità Browser**: Funziona su tutti i principali browser web.

Event loop

L'**event loop** è un meccanismo utilizzato nei sistemi di programmazione asincrona, come Node.js e molti linguaggi di scripting, per gestire eventi e operazioni non bloccanti in modo efficiente.

Ecco una spiegazione semplice e riassuntiva:

1. **Cos'è l'Event Loop?**
 - L'event loop è un ciclo continuo che controlla la coda degli eventi (chiamata anche "queue") e li gestisce uno alla volta.
2. **Cosa fa?**
 - Controlla se ci sono eventi o operazioni da eseguire (come input/output, timer, chiamate di rete).
 - Esegue queste operazioni non appena diventano disponibili, senza bloccare il resto del programma.
3. **Perché è utile?**
 - Permette al programma di gestire molte operazioni simultaneamente senza aspettare che ciascuna sia completata prima di iniziare la successiva.
 - Migliora l'efficienza e la reattività del programma, specialmente nelle applicazioni web.
4. **Come funziona?**
 - **Step 1:** Il programma inizia e l'event loop parte.
 - **Step 2:** Il programma registra (o "submits") delle operazioni asincrone (es. leggere un file, fare una richiesta HTTP).
 - **Step 3:** Quando un'operazione asincrona è pronta (es. la lettura del file è completata), l'event loop la rileva nella coda degli eventi.
 - **Step 4:** L'event loop esegue il callback associato all'evento (una funzione specificata dal programmatore) e poi passa al prossimo evento nella coda.

AJAX

AJAX (Asynchronous JavaScript and XML) è una tecnica utilizzata nelle applicazioni web per aggiornare parti di una pagina web senza doverla ricaricare completamente.

L'**architettura di XMLHttpRequest** è un componente chiave di AJAX che permette ai browser di comunicare con i server in modo asincrono. Ecco una spiegazione semplice e riassuntiva:

1. **Cos'è XMLHttpRequest?**
 - È un oggetto JavaScript utilizzato per inviare e ricevere dati dal server senza dover ricaricare la pagina web.
2. **Come funziona?**
 - **Creazione:** Si crea un'istanza di XMLHttpRequest.
 - **Apertura:** Si configura l'oggetto per il tipo di richiesta (GET o POST) e l'URL del server.

- **Invio:** Si invia la richiesta al server.
 - **Ricezione:** Si imposta una funzione di callback che gestisce la risposta del server una volta ricevuta.
3. **Cosa fa?**
- Invia richieste HTTP al server.
 - Riceve risposte dal server.
 - Permette di aggiornare parti specifiche della pagina web senza ricaricarla.

Architettura di XMLHttpRequest

L'**architettura di XMLHttpRequest** è un componente chiave di AJAX che permette ai browser di comunicare con i server in modo asincrono. Ecco una spiegazione semplice e riassuntiva:

1. **Cos'è XMLHttpRequest?**
 - È un oggetto JavaScript utilizzato per inviare e ricevere dati dal server senza dover ricaricare la pagina web.
2. **Come funziona?**
 - **Creazione:** Si crea un'istanza di XMLHttpRequest.
 - **Apertura:** Si configura l'oggetto per il tipo di richiesta (GET o POST) e l'URL del server.
 - **Invio:** Si invia la richiesta al server.
 - **Ricezione:** Si imposta una funzione di callback che gestisce la risposta del server una volta ricevuta.
3. **Cosa fa?**
 - Invia richieste HTTP al server.
 - Riceve risposte dal server.
 - Permette di aggiornare parti specifiche della pagina web senza ricaricarla.

Promise

1. **Cos'è?**
 - Una Promise è un oggetto che rappresenta un'operazione asincrona e il suo risultato futuro.
2. **Come funziona?**
 - Una Promise può trovarsi in uno di tre stati: *pending* (in attesa), *fulfilled* (completata con successo), o *rejected* (fallita).
 - Quando viene completata, si può ottenere il risultato (o un errore) usando i metodi `.then()` e `.catch()`.
3. **Perché è utile?**
 - Gestisce operazioni asincrone in modo più leggibile rispetto alle callback, migliorando la gestione del codice.


Result

1. **Cos'è?**

- Il "result" è il valore restituito quando una Promise viene risolta con successo.
- Se la Promise viene rifiutata, si ottiene un errore.

```
// URL dell'API per ottenere informazioni su un utente
const url = 'https://jsonplaceholder.typicode.com/users/1';

// Funzione per fare la richiesta usando fetch
function fetchUserData(url) {
  // Restituisce una promessa
  return fetch(url)
    .then(response => {
      // Controlla se la risposta è OK (status 200-299)
      if (!response.ok) {
        // Se non è OK, rifiuta la promessa con un messaggio di errore
        throw new Error(`Network response was not ok: ${response.statusText}`);
      }
      // Restituisce una promessa risolta con i dati JSON
      return response.json();
    })
    .then(data => {
      // Elabora i dati (risultato positivo)
      console.log('User Data:', data);
      return data; // restituisce i dati per ulteriori utilizzi
    })
    .catch(error => {
      // Gestisce gli errori che possono accadere durante la richiesta o l'elaborazione
      console.error('There was a problem with the fetch operation:', error);
      // Puoi restituire un valore di fallback o rilanciare l'errore
      throw error;
    });
}
```



```
// Esegui la funzione per fare la richiesta
fetchUserData(url)
  .then(data => {
    // Ulteriori elaborazioni o utilizzo dei dati ottenuti
    console.log('Further processing with fetched data:', data);
  })
  .catch(error => {
    // Gestione finale dell'errore
    console.error('Final error handling:', error);
  });
```

Microtask

- La microtask queue è una coda speciale che contiene microtask, ossia operazioni asincrone di alta priorità come le callback delle promesse risolte (es. `.then()`, `.catch()`, `.finally()`).

Come funziona?

- Dopo che l'event loop ha eseguito tutti i task principali (es. eventi del DOM, timer), controlla la microtask queue.
- Esegue tutte le microtask nella coda prima di tornare ai task principali.

CORS

CORS (Cross-Origin Resource Sharing) è una politica di sicurezza del web che consente alle pagine web di richiedere risorse da un dominio diverso da quello da cui la pagina stessa è stata caricata

AWAIT ASYNC.

```
// Funzione per fare una richiesta
async function fetchData(url) {
  try {
    const response = await fetch(url);
    if (!response.ok) throw new Error('Errore nella risposta');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

// Chiama la funzione
fetchData('https://api.example.com/data');
```

Frontend

- **Cos'è?:** La parte dell'applicazione che gli utenti vedono e con cui interagiscono.
- **Cosa fa?:** Gestisce l'aspetto della pagina web (design, layout, interfaccia utente).
- **Tecnologie comuni:** HTML, CSS, JavaScript, framework come React, Angular, Vue.js.

Backend

- **Cos'è?:** La parte dell'applicazione che gestisce la logica e i dati sul server.
- **Cosa fa?:** Gestisce l'elaborazione dei dati, l'interazione con il database e la logica dell'applicazione.
- **Tecnologie comuni:** Linguaggi di programmazione come Node.js, Python, Ruby, PHP, Java; database come MySQL, MongoDB.

Siti Dinamici vs API Based

Siti Dinamici

- **Cos'è?:** Siti web che generano contenuti al volo in base alla richiesta dell'utente.
- **Come funziona?:** Utilizzano server-side scripting (come PHP, Python, Ruby) per creare pagine web personalizzate ogni volta che vengono richieste.
- **Esempio:** Un sito di notizie che mostra articoli aggiornati in base ai tuoi interessi o alla tua posizione.

API Based

- **Cos'è?:** Siti web che usano API (Application Programming Interfaces) per ottenere e inviare dati.
- **Come funziona?:** L'interfaccia utente è separata dalla logica di backend. I dati sono recuperati tramite chiamate API da server esterni e visualizzati nel frontend.
- **Esempio:** Un'app meteo che usa un'API per ottenere dati meteorologici e mostrare previsioni sul sito.

Architettura di Node.js

Node.js è un ambiente di esecuzione per JavaScript lato server, che permette di costruire applicazioni web e server. Ecco una spiegazione semplice della sua architettura:

1. **Event-Driven:**
 - **Cos'è?:** Node.js utilizza un modello basato su eventi.
 - **Come funziona?:** Quando si verifica un evento (come una richiesta HTTP), Node.js lo gestisce usando callback (funzioni di ritorno) per rispondere a tali eventi.
2. **Non-Blocking I/O:**
 - **Cos'è?:** Input/Output (I/O) non bloccante.
 - **Come funziona?:** Le operazioni di I/O (come lettura di file o richieste di rete) non bloccano l'esecuzione del codice. Node.js continua a lavorare su altre operazioni mentre aspetta che le operazioni di I/O siano completate.
3. **Single-Threaded Event Loop:**
 - **Cos'è?:** Un ciclo di eventi singolo.
 - **Come funziona?:** Node.js esegue il codice in un unico thread (linea di esecuzione), ma gestisce molte operazioni simultaneamente grazie al suo

ciclo di eventi, che gestisce le richieste e le risposte senza bloccare il thread principale.

4. Modularità:

- **Cos'è?:** Struttura modulare.
- **Come funziona?:** Node.js usa moduli e pacchetti per organizzare il codice e le funzionalità. Puoi importare moduli per utilizzare librerie e funzionalità predefinite

Core Module



Core Module	Description
http	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
querystring	querystring module includes methods to deal with query string.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods, and events to work with file I/O.
util	util module includes utility functions useful for programmers.

Cos'è il Routing?

- **Definizione:** Il routing è il processo di definire e gestire le URL delle richieste per determinare quale codice deve essere eseguito o quale risorsa deve essere restituita.

```
const pathName = req.url;
if (pathName === '/' || pathName === '/home') {
  res.end('Home page');
} else if (pathName === '/contatti') {
  res.end('Contatti');
} else if (pathName === '/info') {
  res.end('Info Page');
} else if (pathName === '/api') {
  res.writeHead(404, {
    'Content-type': 'application/json',
  });
  res.end('Info Page');
} else {
  res.writeHead(404, {
    'Content-type': 'text/html',
  });
  res.end('<h1>404 - Page Not foud</h1>');
}
```