1 Lezione del 28-10-24

1.1 Il modello a oggetti del documento

Il Document Object Model (DOM) è un meccanismo che ci permette di modificare dinamicamente i contenuti di una pagina web attraverso linguaggi di scripting come il JavaScript.

Si ha che, sebbene il codice HTML di una pagina sia effettivamente lineare, la struttura che esso va a definire è fondamentalmente **strutturata**: diversi elementi possono coesistera in relazioni di parentela padre/figlio, fratello/fratello, ecc... Questo significa che si può rappresentare un documento HTML attraverso una **struttura ad albero**, dove ogni elemento viene detto **nodo**.

Le interazioni col DOM avvengono attraverso un oggetto specializzato detto `NodeList` (letteralmente, "lista di nodi"), che mantiene appunto, in maniera simile ad un array, un insieme di nodi. A questo punto, quando si ha accesso al DOM della pagina, modificare un elemento significa trovare il nodo corrispondente in una NodeList, e modificarlo.

1.1.1 Tipi di nodo

Esistono, innanzitutto, alcuni tipi principali di nodi nel DOM:

- **Nodi elemento:** rappresentano elementi veri e propri del documento, come `div`, `a`, `p`, ecc...
- **Nodi testo:** rappresentano il testo contenuto in elementi come paragrafi, anchor, ecc...
- **Nodi attributo:** rappresentano attributi di elementi, come `class`, `id`, `src`, ecc...
- **Nodi commento:** rappresentano commenti nella struttura del documento HTML.

1.1.2 Proprietà fondamentali dei nodi

Un nodo ha, fra le altre, le seguenti proprietà essenziali:

- `childNodes`, una `NodeList` che contiene i figli del nodo;
- `firstChild`, il primo nodo figlio del nodo interessato;
- `lastChild`, come sopra, l'ultimo nodo figlio del nodo interessato;
- `nextSibling`, il prossimo nodo fratello del nodo corrente, cioè il nodo successivo sul livello corrente;
- `nodeName`, il nome del nodo, che equivale al nome del tag per i nodi tag, al nome dell'attributo per i nodi attributo, e a `#text` per i nodi testo;
- `nodeValue`, il valore del nodo, cioè Null per nodi elemento, il testo per nodi testo, e il contenuto dei commenti nei nodi commento;
- `parentNode`: il nodo genitore del nodo corrente;
- `previousSibling`, il precedente nodo fratello del nodo corrente, cioè il nodo precedente sul livello corrente;

- `textContent`, una rappresentazione testuale del nodo, da cui sono stati rimossi i tag.

Per quanto riguarda i nodi elemento, si hanno poi le proprietà:

- `classList`, una lista in sola lettura (con metodi get/set) delle classi CSS assegnate a questo elemento. Le classi, a differenza dello stile, sono da usarsi per la modifica a tempo di esecuzione degli stili, in quanto mantengono più isolate le implementazioni in CSS e in JavaScript, rispettivamente degli stili stessi e della logica che li governa;
- `className`, il nome della classe a cui appartiene l'elemento;
- `id`, l'id dell'elemento;
- `innerHtml`, l'HTML che descrive l'elemento, che può ma **non dovrebbe** essere modificato (vedi attacchi XSS);
- `style`, una lista delle proprietà CSS assegnate a questo elemento, che può influenzare effettivamente gli stessi parametri modificati attraverso un cambio di classe con `classList`, ma il cui utilizzo è sconsigliato in quanto si vanno a creare proprietà via scripting che non hanno riscontro nel CSS;
- `tagName`, il nome del tag che forma l'elemento.

Alcuni tag hanno poi proprietà aggiuntive:

- `href`, indica l'URL collegato nei tag `a`;
- `name`, identifica, in maniera simile a `id`, elementi. Riguarda principalmente campi dati per form: si può infatti usare per `a`, `input`, `textarea` e `form`;
- `src`, indica l'URL sorgente dei contenuti visualizzati da `img`, `input`, `iframe` e `script`;
- `value`, fornisce il valore di input dato dagli elementi di input, come `input`, `textarea` e `submit`.

1.1.3 Accedere al DOM

In JavaScript, si può ottenere un riferimento al DOM attraverso l'oggetto `document`. Questo, oltre al DOM, contiene anche proprietà riguardanti il documento stesso, fra cui l'URL, il tipo di encoding, ecc...

Esistono quindi 3 modi principali per modificare, o comunque interagire, con il DOM:

- **Metodi di selezione**: i metodi di selezione di `document` sono, in ordine di selettività decrescente, `getElementById(id)`, `getElementsByClassName(name)`, e `getElementsByTagName(name)`. Questi cercano, rispettivamente, i nodi per id, per nome di classe, o per nome di tag.

  Esistono poi altri metodi, fra cui ricordiamo `querySelector()` e `querySelectorAll`, che permettono di ottenere elementi nello stesso modo in cui lo faremmo in CSS, cio+ specificando una stringa di query. La differenza fra i due metodi è che `querySelector()` restituisce il primo elemento che rispetta la query, e `querySelectorAll()` li restituisce tutti;

- **Modificazione strutturale dei nodi**: esistono diversi metodi per modificare la struttura del DOM:
  - `appendChild`: inserisce un nuovo nodo figlio in fondo al nodo corrente;
  - `createAttribute`: crea un nuovo nodo attributo sul nodo corrente;
  - `createElement`: crea un nodo elemento;
  - `createTextNode`: crea un nodo di testo;
  - `insertAdjacentElement`: crea un nodo figlio in una di quattro posizioni relative al nodo corrente:
    * `beforebegin`: prima dell'elemento su cui si applica la funzione;
    * `afterbegin`: dentro l'elemento su cui si applica la funzione, come primo figlio;
    * `beforeend`: dentro l'elemento su cui si applica la funzione, come ultimo figlio;
    * `afterend`: dopo l'elemento su cui si applica la funzione.
  - `insertAdjacentText`: come sopra, ma per i nodi testo;
  - `insertBefore`: inserisce un nodo figlio prima di un nodo di riferimento nel nodo corrente;
  - `removeChild`: rimuove un nodo figlio dal nodo corrente;
  - `replaceChild`: rimpiazza un nodo figlio con un figlio diverso.
- **Metodi evento**: il DOM si interfaccia direttamente col gestore di eventi del JavaScript, attraverso il metodo `addEventListener()`. La prossima sezione si interessa appunto di quali funzionalità possono essere implementate attraverso questo sistema.

1.1.4 Temporizzazione del DOM

Il DOM viene caricato al momento di caricamento della pagina. Non dovremmo mai cercare di modificare il DOM prima che questo sia stato caricato, ergo bisogna controllare lo stato di caricamento del DOM prima di accedervi per modifiche. Questo può essere fatto attraverso, come vedremo, il sistema di **eventi** fornito dal JavaScript.

1.2 Gestione degli eventi

In JavaScript è supportato il meccanismo di gestione degli eventi attraverso **event handler**. Un event handler viene definito in codice dal programmatore, e poi registrato attraverso la funzione `addEventLister` di un elemento DOM. A questo punto, a tempo di esecuzione, quando l'evento verrà attivato si avrà che l'event handler verrà eseguito.UN esempio di questo sistema è il seguente:

```
function myHandler() {
  alert("Ahi!");
}

const btn = document.querySelector("#btn");
btn.addEventListener("click", myHandler);
```

In questo caso, al click del pulsante di classe `#btn`, la funzione `myHandler()` verrà invocata e visualizzerà un messaggio a schermo.

Spesso, è più conveniente usare funzioni anonime, o una arrow function, come event handler:

```
const btn = document.getElementById("btn");
btn.addEventListener("click", function() {
  alert("Sono una funzione anonima");
});

const btn = document.getElementById("btn");
btn.addEventListener("click", () => {
  alert("Sono una funzione freccia");
});
```

1.2.1 Tipi di evento

Esistono, oltre al `click`, diversi tipi di eventi. Questi sono:

- **Eventi di mouse**, fra cui:
  - `click`: visto finora, corrisponde al click del mouse su un elemento;
  - `dbclick`: corrisponde a un doppio click su un elemento;
  - `mousedown`: corrisponde alla prima pressione del mouse su un elemento;
  - `mouseup`: corrisponde al rilascio del mouse sull'elemento;
  - `mouseover`: corrisponde a passare (non necessariamente cliccare) con il mouse sopra un elemento;
  - `mouseout`: corrisponde a rimuovere il mouse da un elemento;
  - `mousemove`: corrisponde a muovere il mouse mentre si trova sopra un elemento.
- **Eventi di tastiera**, fra cui:
  - `keydown`: corrisponde alla pressione di un tasto da parte dell'utente;
  - `keyup`: corrisponde al rilascio di un tasto da parte dell'utente.
- **Eventi di tocco**, lanciati da interfacce touchscreen, simili agli eventi lanciati dal mouse;
- **Eventi form**, fra cui:
  - `blur`: lanciato quando quando un elemento di un form perde il focus;
  - `focus`: lanciato quando un elemento di un form ottiene il focus;
  - `change`: lanciato quando una casella di input cambia valore (magari per l'azione dell'utente);
  - `reset`: lanciato al reset del form;
  - `select`: lanciato quando si seleziona del testo;
  - `submit`: lanciato all'invio del form. In particolare, questo evento può essere usato per fare della prevalidazione dei contenuti prima di inviarli al server. Ad esempio:

```
document.querySelector("#loginForm").addEventListener("submit",
    function(e) {
  let pass = document.querySelector("#pw").value;
  if(pass="") {
    alert("Inserisci una password");
    e.preventDefault(); /* questo impedisce il submit chiamato
    dall'utente */
  }
});
```

In generale, l'interazione e la validazione coi formi rimane uno degli usi più popolari del JavaScript. Possiamo infatti usare lo scripting per rispondere alle seguenti casistiche:

  - Movimento fra elementi, di cui possiamo approfittare per dare aiuto contestuale all'utente;
  - Cambiamento di dati di elementi di input, che possiamo usare per fare prevalidazione o come sopra, per dare aiuto contestuale;
  - Invio finale del form, prima del quale, come abbiamo visto, possiamo fare delle operazioni di prevalidazione. Bisogna notare che qualsiasi prevalidazione si faccia sul lato client andrà rifatta sul lato server, in quanto l'ingresso dal lato client non è mai sicuro (il codice inviato da server può essere stato alterato sulla macchina dell'utente). Fare queste validazioni sul lato client resta comunque utile per ridurre il traffico in uscita al server, in quanto in condizoni operative si possono notare errori prima dell'invio dei dati.

- **Eventi media**, relativi a contenuti multimediali, fra cui:
  - `ended`: lanciato quando dell'audio o del video ha finito di eseguire;
  - `pause`: lanciato quando dell'audio o del video è stato messo in pausa;
  - `play`: lanciato quando dell'audio o del video viene avviato;
  - `ratechange`: lanciato quando la velocità di playback di un contenuto multimediale cambia;
  - `volumechange`: lanciato quando il volume di un contenuto multimediale cambia.
- **Eventi frame**, fra cui:
  - `abort`: lanciato quando si arresta il caricamento di una risorsa;
  - `error`: lanciato quando il caricamento di una risorsa va in errore;
  - `load`: lanciato quando la finestra è stata caricata completamente;
  - `DOMContentLoaded`: lanciato quando il documento HTML è stato caricato e il DOM è pronto;
  - `orientationchange`: lanciato quando si cambia orientamento (dello schermo);
  - `resize`: lanciato quando si modifica la dimensione del documento;
  - `scroll`: lanciato quando si scrolla il documento;
  - `unload`: lanciato quando il documento viene rimosso dalla memoria, cioè quando l'utente lascia la pagina.

Gli eventi frame permettono di implementare tecniche di **lazy loading**, o *caricamento pigro*, cioè di caricare contenuti non essenziali della pagina (script, immagini, contenuti multimediali) solo quando l'utente compie determinate azioni (come scorrere sulla pagina, spostarsi fra pagine, ecc...). Così facendo si riduce la cosiddetta **critical rendering path**, cioè il numero minimo di risorse che il browser deve scaricare prima di visualizzare la pagina, e quindi il tempo complessivo di accesso alla pagina.

1.2.2 Caricamento pagina e il DOM

Possiamo quindi risolvere introdotto due paragrafi fa, cioè quello di capire quando la pagina è caricata. Questo può essere verificato attraverso due eventi separati:

- `window.load`: questo evento viene lanciato quando l'intera pagina è caricata (incluse immagini, script e stylesheet). Spesso non è necessario aspettare tanto, in quanto si può semplicemente aspettare il prossimo evento:
- `document.DOMContentLoaded`: questo evento viene lanciato quando il documento HTML è stato completamente scaricato e interpretato. Basta per iniziare a fare modifiche scriptate sul DOM.

Possiamo quindi racchiudere tutto il nostro codice relativo al DOM all'interno di un handler per l'evento `DOMContentLoaded`, ad esempio come:

```
document.addEventListener("DOMContentLoaded", function() {
  /* il mio codice */
});
```

1.2.3 Oggetti evento

Ogni volta che viene lanciato un evento, si va a creare un oggetto relativo a quell'evento, che trasporta informazioni che possono essere raccolte attraverso le proprietà dell'oggetto stesso. Ad esempio, un'evento `click` contiene le proprietà `clientX` e `clientY`, che rappresentano la posizone del cursore dell'utente al momento del click.

Un'evento viene passato come oggetto alle funzioni event handler che prevedono un argomento. Ad esempio si può avere:

```
const menu = document.querySelectorAll("#menu li");
for(let item of menu) {
  item.addEventListener("click", menuHandler);
}
function menuHandler(e) {
  /* e contiene un oggetto dell'evento onclick */
  const x = e.clientX;
  const y = e.clientY;
  /* fai qualcosa con la posizione del cursore x, y */
}
```

1.3 Propagazione di eventi

Quando un'evento viene lanciato su un elemento che ha degli antenati, questo viene propagato sugli antenati. Per ragioni storiche, la propagazione avviene in due direzioni:

- **Fase di cattura, o *"trickle down"*:** nella fase iniziale, il browser propaga l'evento all'elemento più esterno (quindi il tag `html`), e poi via via nei tag interni fino a raggiungere il tag che ha lanciato l'evento in primo luogo. Il browser Netscape ha storicamente adottato questo modello;
- **Fase di bubbling:** nella seconda fase, l'evento viene propagato dall'elemento che lo ha lanciato fino agli elementi esterni. Il browser Explorer ha, di contro, adottato storicamente questo modello.

Oggi, i browser supportano entrambe le fasi, che solitamente svolgono esattamente in quest'ordine. In particolare, si può specificare alla chiamata di `addEventListener` quale fase si vuole usare, col valore default di `bubble`:

```
/* si crea un event listener per la funzione "funzioneCattura" in fase di
    cattura */
elem.addEventListener('click', funzioneCattura, true);
/* oppure */
elem.addEventListener('click', funzioneCattura, { capture: true; });
/* si crea un event listener per la funzione "funzioneBolleggia" in fase
    di bubbling */
elem.addEventListener('click', funzioneBolleggia, false);
/* oppure */
elem.addEventListener('click', funzioneBolleggia, {capture: false; });
/* oppure, visto che e' default */
elem.addEventListener('click', funzioneBolleggia);
```

1.3.1 Fermare la propagazione

Esiste un problema con questo tipo di propagazione degli eventi: a volte vorremmo che un evento venisse catturato, e che poi non si propagasse oltre nella fasse di bubbling. Per fare ciò, possiamo usare il metodo `stopPropagation()` in fase di bubbling, sull'oggetto evento ottenuto dall'event handler. Un'esempio è:

```
const btns = document.querySelectorAll(".plus");
for (let b of btns) {
  b.addEventListener("click", function (e) {
    e.stopPropagation();
    incrementCount(e);
  });
}
```

1.4 Delegazione di eventi

Il meccanismo di propagazione può semplificare la gestione degli eventi nel caso si abbiano più elementi che possono lanciare eventi che devono essere gestiti da un singolo event handler. In questo caso, anziché definire l'event handler su tutti gli elementi, come avevamo fatto nell'esempio precedente:

```
const btns = document.querySelectorAll(".plus");
/* li scorro tutti */
for (let b of btns) {
  b.addEventListener("click", function (e) { /* handler */ });
}
```

si può definire un singolo event handler su un elemento genitore a tutti gli elementi lanciatori di eventi. In questo caso, attraverso la propagazione in fase di bubbling, si ha

che l'evento viene catturato da questo genitore indefferentemente da quale elemento sia a lanciarlo:

```
const parent = document.querySelector("#list");
parent.addEventListener("click", function (e) {
  if(e.target && e.target.nodeName == "BUTTON") {
    /* handler*/
  }
});
```

1.5 Dataset

Esiste una differenza di temporizzazione fra le variabili dichiarate nel codice HTML (e quindi ottenute dal DOM) e quelle dichiarate nel codice JavaScript di un event handler: a tempo di definizione, quest'ultimo non avrà accesso al DOM, e quindi a parametri specifici all'HTML. Per ovviare a questo problema si usano la proprietà `dataset`, e quindi gli attributi `dataset` nei tag HTML: questi sono particolari attributi, prefissi da `data-`, che permettono di inserire informazioni qualsiasi in un elemento HTML. Ad esempio, si potrà avere:

```
<img src="file.png" id="a", data-country="Mozambico">
```

in HTML, e:

```
const element = document.getElementById("a");
const country = element.dataset.country;
/* adesso hai "Mozambico" in country */
```

per accedervi in JavaScript.