Form, validazione e persistenza

Niccolò Maltoni
niccolo.maltoni@diennea.com

Richiamo: DOM ed eventi

Abbiamo già visto che una pagina web è composta da **elementi** (tag HTML) che formano il cosiddetto **DOM** (Document Object Model), ovvero una struttura ad albero che ci dà accesso al contenuto della pagina e ci permette di cambiare o creare qualunque cosa all'interno della stessa.

Gli elementi della pagina possono emettere **eventi** (click, submit, change, ecc.) che possiamo **intercettare** con JavaScript, ad esempio usando `addEventListener()`:

```
button.addEventListener('click', function(event) {
  console.log('Pulsante cliccato!');
});
```

Abbiamo visto, inoltre, che diversi eventi vengono gestiti direttamente dal browser (es. la navigazione al click su un link, o l'invio di un form); è comunque possibile **prevenire** il comportamento predefinito usando il metodo `preventDefault()` dell'oggetto evento passato alla funzione di callback:

```
form.addEventListener('submit', function(event) {
  event.preventDefault();  // Blocca invio tradizionale
});
```

Ma cos'è un form?

Abbiamo più volte accennato all'elemento HTML `<form>`, senza però approfondirlo dovutamente.

Un **form** (letteralmente *modulo*) è un elemento utilizzato per collezionare gli input degli utenti ed inviarli poi ad un server che processerà quelle informazioni (ad esempio, per registrare un nuovo utente, o per effettuare una ricerca):

```html
<form id="loginForm">
  <input id="username" type="text" name="username" placeholder="Username">
  <input id="password" type="password" name="password" placeholder="Password">
  <input id="email" type="email" name="email" placeholder="Email">

  <button type="submit">Login</button>
</form>
```

Al suo interno, un form può contenere *diversi tipi di **input*** (testo, data, select, ecc.) che l'utente può compilare.

Tipicamente contiene anche un **bottone** di tipo `submit` che, quando cliccato, invia i dati del form al server.

Submit del form: `action` e `method`

Un form HTML ha due attributi che controllano come i dati vengono inviati al server:

- **`action`**: URL del server dove inviare i dati (es. `/api/registrazione`)
- **`method`**: tipo di richiesta HTTP:
  - **`GET`**: dati visibili in URL, max ~2KB
  - **`POST`**: dati nel body, size illimitato, per dati sensibili

```html
<!-- Ricerca: GET -->
<form action="/search" method="get">
  <input type="text" name="query">
  <button type="submit">Cerca</button>
</form>
```

```html
<!-- Registrazione: POST -->
<form action="/registrazione" method="post">
  <input type="email" name="email">
  <input type="password" name="password">
  <button type="submit">Registrati</button>
</form>
```

In questa lezione useremo `preventDefault()` per bloccare l'invio tradizionale e gestire i dati con JavaScript.

Input type

Tramite l'attributo `type` di un elemento `<input>` possiamo specificare il tipo di dato che vogliamo ricevere dall'utente, e il browser ci fornirà un'interfaccia adatta per quel tipo di dato.

`type="text"`: testo generico, su un'unica riga

```
<input type="text"  name="nome" placeholder="Il tuo nome">
```

`type="password"`: testo nascosto (per password):

```
<input type="password" name="pwd" placeholder="Password">
```

`type="email"`: testo con validazione email

```
<input type="email" name="email" placeholder="email@example.com">
```

`type="number"`: input per numeri, con frecce +/-

```
<input type="number" name="eta" min="18" max="99" placeholder="Età">
```

`type="date"`: selettore per date, con calendario

```
<input type="date" name="nascita">
```

`type="file"`: input per file (upload)

```
<input type="file" name="foto">
```

`type="checkbox"`: spunta (per selezioni multiple)

```
<input type="checkbox" name="newsletter" value="si">
```

`type="radio"`: spunta (per selezioni singole)

```
<input type="radio" id="male" name="gender" value="male">
<input type="radio" id="female" name="gender" value="female">
```

Come sempre, ci sono molti altri tipi di input che possiamo esplorare nella documentazione MDN.

Leggere i dati del form: accesso diretto

Il primo approccio è leggere i campi **uno per uno** con `id` o `name`, usando `.value`.

```
const form = document.getElementById('loginForm');
const emailInput = document.getElementById('email');
const passwordInput = document.getElementById('password');

form.addEventListener('submit', function(event) {
  event.preventDefault();

  const email = emailInput.value;
  const password = passwordInput.value;

  console.log({ email, password });
});
```

Questo approccio è semplice e chiaro, ma diventa molto verboso se il form ha molti campi.

Generalmente è preferibile evitare questo metodo salvo necessità specifiche (es. campi dinamici senza `name`).

Leggere i dati del form: `document.forms`

I form del documento sono esposti dal DOM in `document.forms`:

index.html

```html
<form name="loginForm">
  <input name="email" type="email">
</form>
```

script.js

```js
const form = document.forms.loginForm;
// potremmo anche usare l'indice: const form = document.forms[0];
const email = form.elements.email.value;
```

Si tratta di una collezione contenente tutti i form della pagina, accessibili tramite `name` o indice; è infatti una cosiddetta “named collection”, sia **associativa** che **ordinata**.

Ciascun oggetto form ha a sua volta una collezione `elements` che contiene tutti i campi del form, anch’essa accessibile tramite `name` o indice.

Se più campi condividono lo stesso `name`, `form.elements[name]` restituisce una collezione:

index.html

```html
<form>
  <input name="interessi" type="checkbox" value="sport">
  <input name="interessi" type="checkbox" value="musica">
</form>
```

script.js

```js
const form = document.forms[0];
const elems = form.elements.interessi;
console.log(elems[0]); // [object HTMLInputElement]
```

Fieldset

Un form può avere uno o più elementi `<fieldset>` all'interno.

Questi elementi si comportano come dei "sub-form", e hanno una proprietà `elements` per accedere agli elementi al loro interno:

index.html

```html
<form id="form">
  <fieldset name="userFields">
    <legend>info</legend>
    <input name="login" type="text">
  </fieldset>
</form>
```

script.js

```js
console.log(form.elements.login); // <input name="login">

const fieldset = form.elements.userFields;
console.log(fieldset); // HTMLFieldSetElement

// possiamo ottenere l'input sia dal nome del form sia dal fieldset
console.log(fieldset.elements.login == form.elements.login); // true
```

Leggere i dati del form: `FormData`

Il metodo più conciso per accedere a un form è usando l'oggetto **`FormData`**.

```
const form = document.querySelector('#loginForm');

form.addEventListener('submit', function(event) {
  event.preventDefault();

  // Crea FormData dal form
  const formData = new FormData(this); // in questo caso this === form

  // Convertire a oggetto JavaScript
  const dati = Object.fromEntries(formData);
  console.log(dati);  // { username: 'mario', password: '123', email: 'mario@example.com' }
});
```

Il costruttore `FormData` accetta un elemento form e legge **automaticamente** tutti i campi con `name`, creando un oggetto dedicato.

È particolarmente utile per inviare dati a un server, in quanto i metodi per le richieste di rete, come ad esempio `fetch`, possono accettare gli oggetti `FormData` come body della richiesta, ma è anche un modo rapido per ottenere un oggetto con tutti i dati del form senza dover accedere a ciascun campo manualmente.

Esercizio: FormData in azione

Leggi i dati di un form

Crea un form HTML con almeno 3 campi di input (ad esempio: nome, email, messaggio).

Scrivi JavaScript che:

1. Intercetta l'evento di submit del form
2. Blocca il comportamento predefinito
3. Estrae i dati del form usando `FormData`
4. Stampa l'oggetto risultante in console

Verifica: compila il form e clicca il bottone. In console dovrai vedere un oggetto con le proprietà corrispondenti ai campi compilati.

Input di selezione: `checkbox`, `radio`, `select`

A differenza degli input di testo, i campi di selezione richiedono un trattamento speciale per leggere i valori:

Checkbox:

```html
<label>
  <input type="checkbox" name="newsletter" value="si">
  Iscriviti alla newsletter
</label>
```

```js
// Checkbox/radio con .checked
const newsletter = document.querySelector('[name="newsletter"]');
console.log(newsletter.checked); // true/false
```

Radio (selezione esclusiva):

```html
<label>
  <input type="radio" name="lingua" value="it"> Italiano
</label>
<label>
  <input type="radio" name="lingua" value="en"> English
</label>
```

```js
const checked = document.querySelector('[name="lingua"]:checked');
console.log(checked?.value); // 'it'/'en'/undefined
```

Select (menu a tendina):

```html
<select name="colore">
  <option value="">Seleziona</option>
  <option value="rosso">Rosso</option>
  <option value="blu">Blu</option>
</select>
```

```js
const select = document.querySelector('[name="colore"]');
console.log(select.value); // 'rosso', 'blu' o ''
```

Tramite `FormData`, invece, questi campi vengono gestiti automaticamente, restituendo i valori selezionati.

Validazione lato client: perché?

La **validazione lato client** (nel browser) migliora l'**esperienza utente**:

- **Feedback immediato**: l'utente sa subito se ha sbagliato
- **Riduce richieste server**: evita invii inutili
- **Migliore UX**: errori chiari prima dell'invio

> ⚠ **Attenzione**
>
> Bisogna comunque tenere a mente che la validazione client-side **non è sicura**! Devi **sempre** validare anche sul server (backend).
>
> Un utente esperto può bypassare la validazione JavaScript. La sicurezza vera è sul server.

Validazione HTML5: attributi

HTML5 fornisce **attributi di validazione** integrati:

```html
<form id="regForm">
  <input type="text" name="username"
    required
    minlength="3"
    maxlength="20"
    placeholder="Username (3-20 caratteri)">

  <input type="email" name="email"
    required
    placeholder="Email obbligatoria">

  <input type="number" name="eta"
    required
    min="18"
    max="99"
    placeholder="Età (18-99)">

  <input type="password" name="password"
    required
    pattern="[A-Za-z0-9]{6,}"
    placeholder="Password (min 6 caratteri alfanumerici)">

  <button type="submit">Registrati</button>
</form>
```

Attributi di validazione HTML5

Attributo	Descrizione	Esempio
`required`	Campo obbligatorio	`<input required>`
`minlength`	Lunghezza minima	`<input minlength="3">`
`maxlength`	Lunghezza massima	`<input maxlength="20">`
`min`	Valore minimo (number, date)	`<input type="number" min="18">`
`max`	Valore massimo (number, date)	`<input type="number" max="120">`
`pattern`	Espressione regolare (regex)	`<input pattern="[0-9]{5}">`

Il browser mostra **messaggi di errore automatici** quando si prova a inviare un form non valido.

Non approfondiremo le regex qui. Se siete interessati, regex101.com e MDN sono risorse eccellenti.

Validazione: HTML5 vs JavaScript

Si tratta di due approcci complementari per validare i form:

Validazione HTML5 (dichiarativa):

```html
<form id="regForm">
  <input type="text" name="username"
    required
    minlength="3"
    maxlength="20">

  <input type="email" name="email" required>

  <button type="submit">
    Registrati
  </button>
</form>
```

La soluzione più semplice, con messaggi automatici.

Validazione JavaScript (programmatica):

```javascript
form.addEventListener('submit', function(e) {
    e.preventDefault();

    const username = form.username.value.trim();

    if (username.length < 3) {
      alert('Username troppo corto');
      return;
    }

    console.log('Form valido!');
});
```

Permette logica complessa e messaggi personalizzati, ma richiede più codice.

Generalmente, è consigliato utilizzare HTML5 solamente per la validazione base.

JavaScript è da preferirsi per tutte le validazioni un minimo complesse.

Eventi utili per validazione

Oltre a `submit`, possiamo intercettare altri eventi utili per una validazione real-time.

Evento	Quando scatta	Utilizzo pratico
`focus`	Campo selezionato	Suggerimenti, evidenziazione
`blur`	Focus rimosso dal campo	Messaggi (i.e. “Campo vuoto”)
`input`	Ad ogni carattere digitato	Feedback in tempo reale
`change`	Focus rimosso dal campo dopo aver modificato il valore	Validazione al completamento

Esercizio: validazione HTML5

Form con validazione HTML5

Crea un form di registrazione con:

1. Nome: obbligatorio, lunghezza 3-30 caratteri
2. Email: obbligatoria, formato email
3. Numero di telefono: non obbligatorio, pattern di 10 cifre numeriche, placeholder “10 cifre”
4. Bottone “Registrati”

Prova a inviare il form senza compilarlo, con email non valida, con numero incompleto. Osserva i messaggi di errore automatici del browser.

Form con validazione in JavaScript

Riscrivi lo stesso form, ma questa volta senza usare gli attributi di validazione HTML5. Implementa la validazione usando JavaScript.

Persistenza dei dati

La compilazione di un form è un'azione che spesso vogliamo memorizzare per l'utente.

Tuttavia, non possiamo semplicemente salvare i dati in variabili JavaScript, perché quando la pagina viene ricaricata, lo stato del programma viene azzerato.

Il browser offre diverse soluzioni per la **persistenza dei dati**:

- **Cookie**: piccoli file di testo inviati al server ad ogni richiesta
- **Web Storage**: API per salvare dati localmente nel browser (`localStorage` e `sessionStorage`)
- **IndexedDB**: database NoSQL per grandi quantità di dati strutturati

Cookie

I cookie sono stati storicamente il metodo principale per la persistenza dei dati nel browser. Si tratta di piccole stringhe di dati, memorizzate direttamente nel browser, che vengono inviate al server *ad **ogni** richiesta HTTP*.

```
// Scrivere un cookie
document.cookie = 'username=Mario';
document.cookie = 'theme=dark; path=/; max-age=3600';  // Scade tra 1 ora

// Leggere tutti i cookie (stringa unica!)
console.log(document.cookie);  // 'username=Mario; theme=dark'
```

Limitazioni:

- API scomoda (tutto in una stringa, no metodi `get`/`set`)
- Massimo **4KB** per cookie
- Inviati al server ad **ogni** richiesta (overhead di rete)
- Principalmente usati per **autenticazione** lato server

Web Storage API

Il browser fornisce API moderne e semplici per salvare dati localmente, dette **Web Storage**. Sono disponibili due storage con interfaccia identica ma durata diversa:

- `localStorage`: dati persistono indefinitamente
- `sessionStorage`: dati scompaiono al riavvio browser (o chiusura tab)

Entrambi offrono la stessa API:

```
const storage = localStorage;  // oppure sessionStorage

// Salvare
storage.setItem('chiave', 'valore');

// Leggere
const valore = storage.getItem('chiave');

// Eliminare
storage.removeItem('chiave');
storage.clear();  // Cancella tutto
```

Limite: ~5-10 MB per dominio, dati salvati come **stringhe**.

LocalStorage

`localStorage` permette di salvare dati che persistono indefinitamente nel browser:

```
// Salvare preferenze
localStorage.setItem('tema', 'dark');
localStorage.setItem('lingua', 'it');

// Leggere
const tema = localStorage.getItem('tema');
console.log('Tema salvato:', tema);  // 'dark'

// Controllare se esiste
if (localStorage.getItem('tema')) {
  console.log('Tema trovato!');
}

// Rimuovere e pulire
localStorage.removeItem('tema');
localStorage.clear();  // Cancella tutto
```

Prova ad aprire la **Console** dei DevTools e scrivere questi comandi!

I dati rimangono anche se ricarichi la pagina (F5).

localStorage vs sessionStorage

Due API identiche, ma con **durata diversa**:

Caratteristica	localStorage	sessionStorage
Durata	Indefinita (finché non cancellata)	Fino alla chiusura del tab
Condivisione	Condivisa tra tutti i tab dello stesso sito	Isolata per ogni tab
Uso tipico	Preferenze utente, temi, cache	Dati temporanei, sessione

L'API è **esattamente la stessa** (`setItem`, `getItem`, `removeItem`, `clear`).

> ⚠️ **Limitazioni di sicurezza**
>
> **Non salvare dati sensibili** (password, token, carte di credito) in `localStorage`!
>
> È vulnerabile ad attacchi XSS: qualsiasi script sulla pagina può leggere i dati.

Esempio: salvare dati del form

Salviamo i dati del form in localStorage per recuperarli al prossimo caricamento:

```
const form = document.querySelector('#profiloForm');

// Al caricamento, recupera dati salvati
window.addEventListener('DOMContentLoaded', function() {
  const nomeSalvato = localStorage.getItem('nome');
  const emailSalvata = localStorage.getItem('email');

  if (nomeSalvato) form.nome.value = nomeSalvato;
  if (emailSalvata) form.email.value = emailSalvata;
});

// Al submit, salva dati
form.addEventListener('submit', function(event) {
  event.preventDefault();

  localStorage.setItem('nome', form.nome.value);
  localStorage.setItem('email', form.email.value);

  alert('Profilo salvato!');
});
```

Ora i dati rimangono anche dopo la ricarica della pagina!

Esercizio finale: aggiungi form al ricettario

Crea ricette con tre form incrementali

Riprendiamo il ricettario della **lezione 4**. Aggiungi tre form separati all'interno della card:

1. **Ricetta base** (nome, immagine, descrizione + pulsante "Crea ricetta") — **alla fine della card, dopo la navigazione**;

Al submit: crea istanza `Ricetta`, aggiungila a `ricettario`, visualizza sulla pagina

2. **Ingredienti** (quantità opzionale + nome + pulsante "Aggiungi");

Al submit: aggiunge ingrediente alla ricetta corrente, refresha la lista

3. **Preparazione** (textarea passo + pulsante "Aggiungi");

Al submit: aggiunge passo alla ricetta corrente, refresha la lista

Ogni form si svuota dopo submit. Flusso: crea ricetta → aggiungi ingredienti/passaggi uno per uno → ripeti per altre ricette. Precedi il form base con un titolo `<h3>` e disponi gli input su più righe con `<div>` per chiarezza.

Esempio: ricettario con form

Il risultato atteso assomiglia a questo: