# Manuale HTML e JavaScript

V1.09

**Disclaimer**: Questo manuale è in continua evoluzione e verrà aggiornato progressivamente con nuovi contenuti e correzioni man mano che se ne presenterà la necessità. Si ricorda che la **principale risorsa per la programmazione resta la documentazione ufficiale**; questo documento serve solo come raccolta di argomenti complessi, arricchiti da alcuni esempi.

Il manuale inoltre include la documentazione all’ambiente semplificato sviluppato dal sottoscritto i cui componenti comuni verranno tipicamente (**ma non sempre**) forniti.

Il manuale è suddiviso in due parti:

1. **Manuale dei Componenti UI**: Fornisce descrizioni ed esempi su come progettare interfacce utente utilizzando i fogli di stile CSS riuniti nel file *common.css*. Questo file è stato creato dal sottoscritto per garantire una coerenza visiva nelle nostre applicazioni in ambito scolastico.
2. **Manuale JavaScript**: Contiene le principali linee guida ed esempi relativi alla dichiarazione di variabili, funzioni, classi, proprietà, chiamate asincrone, cicli, e alla manipolazione del DOM in tempo reale. Contiene documentazione su metodi di utilità sviluppati che sono tipicamente condivisi tra i vari progetti.

Sommario

[Manuale HTML e JavaScript 1](#_Toc197852129)

[Manuale dei componenti UI 3](#_Toc197852130)

[Classe .container 3](#_Toc197852131)

[Classe .cols 3](#_Toc197852132)

[Classe .left, .right, .center 4](#_Toc197852133)

[HTML Form 4](#_Toc197852134)

[Tag <form> 5](#_Toc197852135)

[Number input field 5](#_Toc197852136)

[Text input field 6](#_Toc197852137)

[Date input field 6](#_Toc197852138)

[DateTime input field 6](#_Toc197852139)

[Email input field 7](#_Toc197852140)

[Password input field 7](#_Toc197852141)

[TextArea input field 8](#_Toc197852142)

[Checkbox 8](#_Toc197852143)

[Switch 8](#_Toc197852144)

[Radio button 9](#_Toc197852145)

[Select field 9](#_Toc197852146)

[Fieldset 10](#_Toc197852147)

[Bottoni 10](#_Toc197852148)

[Tabelle 12](#_Toc197852149)

[Manuale JavaScript 14](#_Toc197852150)

[Inclusione del file JavaScript 14](#_Toc197852151)

[Dichiarazione variabili e costanti 14](#_Toc197852152)

[Strict mode 15](#_Toc197852153)

[Funzioni 15](#_Toc197852154)

[Array 16](#_Toc197852155)

[Accesso agli elementi del DOM 17](#_Toc197852156)

[Accesso e modifica alle proprietà degli elementi del DOM 18](#_Toc197852157)

[Validazioni 20](#_Toc197852158)

[Classi 20](#_Toc197852159)

[Repository (Item Model) 23](#_Toc197852160)

[Fetch API 25](#_Toc197852161)

[Custom Events 27](#_Toc197852162)

[Promise 30](#_Toc197852163)

[Web API Manager 31](#_Toc197852164)

[Popup 32](#_Toc197852165)

[Progressive Web App 32](#_Toc197852166)

[Web Components 32](#_Toc197852167)

## Manuale dei componenti UI

### Classe .container

Crea un box bianco dai bordi arrotondati. Viene utilizzato per contenere al suo interno altri elementi, che verranno disposti incolonnati. Tipicamente utilizzato su un <form>.

Immagine che contiene schermata, Rettangolo, blu, Blu elettrico

Descrizione generata automaticamente

<form class="container">

[altri elementi]

</form>

### Classe .cols

**Grid system**, dispone gli elementi collocati all’interno di un elemento di classe .cols su singola riga, ripartendo proporzionalmente lo spazio disponibile su ciascun elemento figlio: se i figli sono due, ognuno avrà il 50% dello spazio, se sono 3 sarà il 33% e così via.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

<div class="cols">

[primo elemento]

[secondo elemento]

[terzo elemento]

[quarto elemento]

[quinto elemento]

</div>

### Classe .left, .right, .center

Se usato all’interno di un contenitore di classe .cols o tag <footer> o <header>, dispone l’elemento a destra, sinistra o in mezzo rispetto allo spazio disponibile

<div class="footer">

<div class="left">

  <button type="button" id="btnBack" icon="back">Indietro</button>

  </div>

  <div class="right">

  <button type="button" id="btnAnnulla" icon="refresh">Annulla</button>

     <button type="button" id="btnElimina" icon="delete" color="red">Elimina</button>

</div>

</div>

### HTML Form

Le HTML form (form HTML) sono elementi di una pagina web che **consentono agli utenti di inserire dati e inviarli a un server per l'elaborazione**.

Un form HTML è definito con il tag **<form>**, e può contenere diversi tipi di campi di input come:

<input type="text"> campi di testo per inserire testo generico.

<input type="password"> campi di testo per inserire password.

<input type="email"> campi di testo per inserire un indirizzo email.

<input type="number"> campi di testo per inserire numeri.

<textarea> aree di testo per inserire testi lunghi.

<input type="checkbox"> checkbox per selezioni multiple.

<input type="radio"> radio button per selezioni singole.

<select> menu a tendina per scelte multiple o singole.

#### Tag <form>

**Contenitore** per gli oggetti di acquisizione dati (input field, select, radio, ecc): è **obbligatorio** che questi elementi siano **all’interno di un contenitore di tipo <form>**.

Per questo motivo, viene **tipicamente** inserito come primo figlio del body, al quale viene applicata la classe **.container**.

Immagine che contiene testo, schermata, design

Descrizione generata automaticamente

<form class="container">

[altri input/elementi]

</form>

#### Number input field

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

<div class="input-field">

<label for="">LABEL TEXT</label>

<input type="number" id="" min="0" max="100" value="0" placeholder="PLACEHOLDER TEXT"/>

</div>

È opzionalmente possibile specificare, sul tag di **<input>** l’attributo **required** per apportare un indicatore visivo di obbligatorietà. Utilizzando l’attributo **disabled**, invece, il field è visualizzato in sola lettura.

#### Text input field

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

<div class="input-field">

<label for="">LABEL TEXT</label>

<input type="text" id="" placeholder="PLACEHOLDER TEXT"/>

</div>

È opzionalmente possibile specificare, sul tag di **<input>** l’attributo **required** per apportare un indicatore visivo di obbligatorietà. Utilizzando l’attributo **disabled**, invece, il field è visualizzato in sola lettura.

#### Date input field

Immagine che contiene testo, schermata, Carattere, bianco

Descrizione generata automaticamente

<div class="input-field">

<label for="">LABEL TEXT</label>

<input type="date" id=""/>

</div>

È opzionalmente possibile specificare, sul tag di **<input>** l’attributo **required** per apportare un indicatore visivo di obbligatorietà. Utilizzando l’attributo **disabled**, invece, il field è visualizzato in sola lettura.

#### DateTime input field

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

<div class="input-field">

<label for="">LABEL TEXT</label>

<input type="datetime-local" id=""/>

</div>

È opzionalmente possibile specificare, sul tag di **<input>** l’attributo **required** per apportare un indicatore visivo di obbligatorietà. Utilizzando l’attributo **disabled**, invece, il field è visualizzato in sola lettura.

#### Email input field

Immagine che contiene testo, schermata, Carattere, bianco

Descrizione generata automaticamente

<div class="input-field">

<label for="">LABEL TEXT</label>

<input type="email" id="" placeholder="PLACEHOLDER TEXT"/>

</div>

È opzionalmente possibile specificare, sul tag di **<input>** l’attributo **required** per apportare un indicatore visivo di obbligatorietà. Utilizzando l’attributo **disabled**, invece, il field è visualizzato in sola lettura.

#### Password input field

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

<div class="input-field">

<label for="">LABEL TEXT</label>

<input type="password" id="" minlength="8" placeholder="PLACEHOLDER TEXT"/>

</div>

È opzionalmente possibile specificare, sul tag di **<input>** l’attributo **required** per apportare un indicatore visivo di obbligatorietà. Utilizzando l’attributo **disabled**, invece, il field è visualizzato in sola lettura.

#### TextArea input field

Immagine che contiene ricevuta, bianco, testo, algebra

Descrizione generata automaticamente

<div class="input-field">

<label for="">LABEL TEXT</label>

<textarea rows="4" id="" placeholder="PLACEHOLDER TEXT"/></textarea>

</div>

È opzionalmente possibile specificare, sul tag di **<input>** l’attributo **required** per apportare un indicatore visivo di obbligatorietà. Utilizzando l’attributo **disabled**, invece, il field è visualizzato in sola lettura.

#### Checkbox

Immagine che contiene testo, Carattere, bianco, design

Descrizione generata automaticamente

<div class="checkbox-field">

<label for="">LABEL TEXT</label>

<input type="checkbox" id=""/>

</div>

È opzionalmente possibile specificare, sul tag di **<input>** l’attributo **required** per apportare un indicatore visivo di obbligatorietà. Utilizzando l’attributo **disabled**, invece, il field è visualizzato in sola lettura.

#### Switch



<div class="switch-field">

<label for="">LABEL TEXT</label>

<input type="checkbox" id=""/>

</div>

È opzionalmente possibile specificare, sul tag di **<input>** l’attributo **required** per apportare un indicatore visivo di obbligatorietà. Utilizzando l’attributo **disabled**, invece, il field è visualizzato in sola lettura.

#### Radio button



<div class="radio-field">

<label for="">LABEL TEXT</label>

      <input type="radio" id="" name="" checked/>

</div>

Se più radio condividono lo stesso valore dell’attributo name essi entrano a far parte dello stesso gruppo dove solo una delle opzioni può essere checked.

È opzionalmente possibile specificare l’attributo **required** per apportare un indicatore visivo di obbligatorietà.

#### Select field

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

<div class="select-field">

<label for="">Marchio</label>

<select id="">

<option value="" selected>OPTION 1 TEXT</option>

<option value="">OPTION 2 TEXT</option>

<option value="">OPTION 3 TEXT</option>

<option value="">OPTION 4 TEXT</option>

</select>

</div>

È opzionalmente possibile specificare, sul tag di **<input>** l’attributo **required** per apportare un indicatore visivo di obbligatorietà. Utilizzando l’attributo **disabled**, invece, il field è visualizzato in sola lettura.

#### Fieldset

Immagine che contiene testo, Carattere, numero, schermata

Descrizione generata automaticamente

<fieldset id="">

<legend>LEGEND TEXT</legend>

       […]

</fieldset>

È opzionalmente possibile specificare, sul tag di **<fieldset>** l’attributo **disabled**, per rendere tutti i campi in sola lettura.

### Bottoni

Elemento cliccabile con label ed icona. L’icona viene specificata all’interno dell’attributo icon e riferisce ad una specifica icona disponibile nella cartella icons. È possibile impostare varie combinazioni di assenza o presenza di testo e/o icona per ottenere l’effetto visualizzato in immagine.

I bottoni sono collocati **tipicamente** all’interno del tag **<form>** ma possono essere utilizzati anche al di fuori di esso.



<button type="button" icon="" id="" color="" onclick="">LABEL TEXT</button>

Ogni bottone prevede a sua volta la possibilità di definire il colore di sfondo utilizzando l’attributo **color** che riferisce ai colori disponibili nel foglio di stile *color.css* che ne personalizzano ulteriormente l’aspetto che, come tutte le classi CSS, possono anche essere **combinate**.

* nobg: utilizzata per mostrare un bottone senza background color;
* small: mostra un bottone di 40x40 pixel con label in overlay, definito tramite l’attributo **hint** e **hint-position**=”*left|right*”;

Anche per ciò che concerne l’icona, sono disponibili una serie di [icone predefinite](https://fonts.google.com/icons) che vanno specificate nell’attributo **icon**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Icone** | | | | |
| add | bar\_chart | calculator | camera | cart |
| cart\_add | copy | credit\_card | cut | database |
| delete | description | donut\_chart | edit | euro |
| events | favorite | folder | grid | groups |
| image | info | keep | list | location |
| mail | map | my\_location | payments | pie\_chart |
| receipt | recycling | remove | rocket | save |
| savings | sell | settings | skull | shop |
| star | today | tune | visibility | visibility\_off |
| undo | redo | back | forward | search |
| box | order | shipment | warehouse | user |
| close | check | home | refresh | shopping\_bag |
| print | download | upload |  |  |

L’utilizzo del valore *google* nell’attributo **icon** renderizza il pulsante di login tramite Google.

<button type="button" icon="google" id=""></button>



### Tabelle

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Le tabelle sono lo strumento più comune di presentazione di informazioni appartenenti ad oggetti dello stesso tipo o di tipi compatibili. Le tabelle offerte dalla libreria di supporto creata offrono alcune piccole possibilità di personalizzazione.

La struttura della tabella è generalmente creata **vuota** e successivamente riempita con del codice JavaScript **che ne va a modificare il DOM**, aggiungendo/modificando/rimuovendo gli elementi esistenti, come indicato nella sezione Accesso e modifica alle proprietà degli elementi del DOM.

La struttura di HTML di base della tabella è la seguente:

<!-- Tabella persone -->

<table id="tblPersone">

<thead>

<tr>

    <th>Codice Fiscale</th>

      <th>Nome</th>

      <th>Cognome</th>

    </tr>

  </thead>

  <tbody>

  <tr>

    <td colspan="3">Nessun dato</td> <!— Da visualizzare quando vuota-->

    </tr>

  </tbody>

</table>

Il codice appena presentato visualizza una tabella che si espande in verticale occupando lo spazio disponibile, visualizzando una scrollbar che permette di scorrere il suo contenuto qualora non sia totalmente visibile.

Un secondo effetto di scorrimento verticale si può ottenere utilizzando gli **sticky header**. Lo sticky header è una impostazione dello header della tabella che scrolla insieme al contenuto scrollabile, in modo da essere sempre visibile.

Per ottenere una tabella con **sticky header**, è sufficiente wrapparla intorno ad un elemento div, come per esempio

<div class="cols">

<!-- Tabella persone -->

<table id="tblPersone">

[…]

</table>

</div>

Una tabella con sticky header ha una altezza massima prefissata di 300px. Qualora l’impostazione volesse essere modificata, procedere con la definizione di una regola CSS di sovrascrittura del selettore.

div:has(> table) {

    max-height: 300px;

    overflow-y: auto;

}

L’utilizzo dell’attributo color sulla definizione di un tag <tr> o <td> comperterà l’applicazione del colore di sfondo a tutta la riga o alla sola cella

TODO DEFINIZIONE WIDTH COLONNE

## Manuale JavaScript

### Inclusione del file JavaScript

L’inclusione del file JavaScript nella propria pagina web avviene in maniera molto simile a quello che accade con i file Css.

Per ogni file .js da utilizzare, utilizzare il tag script, specificando nel suo attributo src la posizione (**generalmente relativa**) dell’elemento.

<script src="./path\_to\_file.js"></script>

**NB:** se il vostro progetto utilizza più file .js, importarli in ordine di utilizzo.

### Dichiarazione variabili e costanti

Per la dichiarazione di variabili si usano rispettivamente le parole chiave **let** e **const**.

* **let** viene usata per la dichiarazione di variabili, ovvero elementi ai quali è possibile ri-assegnare un valore

let mediavoti = 6.5;

* **const** viene usata per la dichiarazione di costanti, ovvero elementi ai quali non è possibile ri-assegnare un valore

const aliquota = 10;

Ricordiamo inoltre che JavaScript è un linguaggio di programmazione a **tipizzazione dinamica e debole**, pertanto prestiamo attenzione al fatto che:

1. Il tipo della variabile non è dichiarato esplicitamente ma è assegnato in base al suo valore. Il tipo della variabile può cambiare a runtime, questo vuol dire che una variabile che contiene il valore 10 è intera tuttavia successivamente possiamo impostarci la stringa “ciao” ed essa diventerà di tipo stringa. Evitiamo di fare queste cose – **tipizzazione dinamica**
2. JavaScript consente di eseguire operazioni su variabili di tipi diversi senza richiedere conversioni esplicite, ma questo può portare a risultati inattesi – **tipizzazione debole**

**Attenzione!**

JavaScript è un linguaggio di programmazione **case-sensitive**, ciò significa che distingue tra lettere maiuscole e minuscole nei nomi di variabili, funzioni, proprietà, parole chiave e altro.

Di conseguenza, **myVariable, MyVariable, e myvariable** sono considerati nomi di variabili distinti in JavaScript

### Strict mode

Per rendere il linguaggio più rigoroso e limitare la flessibilità volutamente offerta da JavaScript noi utilizziamo la strict mode. La strict mode in JavaScript è una modalità che rende il linguaggio più rigoroso, aiutando a evitare errori comuni e a scrivere codice più sicuro e prevedibile.

Quando si attiva la strict mode aggiungendo "use strict"; all'inizio di uno script o di una funzione, vengono imposti diversi vincoli

// Whole-script strict mode syntax

"use strict";

### Funzioni

Le funzioni in JavaScript sono blocchi di codice riutilizzabili che eseguono un insieme di istruzioni. Possono accettare **opzionalmente** parametri in ingresso, eseguire operazioni e restituire un valore in uscita. Le funzioni sono fondamentali per organizzare e modularizzare il codice.

Le funzioni vanno dichiarate attraverso la parola chiave function che è seguita dal nome della stessa e, tra parentesi tonde, gli eventuali parametri **separati da virgola** ai quali non è necessario specificarne il tipo (per via della tipizzazione dinamica e debole descritta in precedenza). Se non ci sono parametri in ingresso le parentesi tonde non avranno contenuto.

La funzione inoltre può prevedere **un** valore di ritorno.

/\*\*

 \* Dato un importo e una percentuale calcola il valore

 \*/

function calcolaPercentuale(importo, percentuale) {

    let valore = (importo \* percentuale) / 100;

    return Number(valore);

}

### Array

Gli array in JavaScript sono oggetti specializzati che permettono di **memorizzare più valori in una singola variabile**. Questi valori sono **indicizzati** e possono essere di qualsiasi tipo di dato, inclusi numeri, stringhe, oggetti e persino altri array.

* **Creazione**: l’array può essere creato utilizzando la notazione tra parentesi quadre o quella orientata agli oggetti (equivalenti)

// Creazione di array vuoti

let nomi = [];

let cognomi = new Array();

// Creazione e pre assegnazione di valori

let numeri = [1, 2, 3, 4, 5];

let colori = ["rosso", "verde", "blu"];

* **Accesso** agli elementi: Gli elementi di un array sono accessibili tramite il loro indice che parte da 0

let colore = colori[1]; // La variabile colore vale “verde”

* **Lunghezza** dell’array: la proprietà length permette di conoscere quanti elementi sono memorizzati nell’array. Ricordate che un array per esempio di lunghezza 5 contiene elementi che vanno da indice 0 a 4.

function getRadioValue(radioName) {

    let i;

    /\* Ottengo tutti I radio con lo stesso nome che quindi sono raggruppati \*/

    let radioElements = document.getElementsByName(radioName);

/\* Itero su tutti gli elementi acquisiti accedendo all’array radioElements \*/

    for (i = 0; i < radioElements.length; i++) {

        if (radioElements[i].checked) {

            return Number(radioElements[i].value);

        }

    }

}

* **Aggiunta** di nuovi elementi attraverso la funzione push()

let numeri = [1, 2, 3];

numeri.push(4, 5);

console.log(numeri); // Output: [1, 2, 3, 4, 5]

* **Rimozione** di elementi attraverso la funzione pop()

let numeri = [1, 2, 3, 4];

let ultimo = numeri.pop();

console.log(ultimo); // Output: 4

console.log(numeri); // Output: [1, 2, 3]

* **Unione** di due array attraverso la funzione concat()

let numeri1 = [1, 2, 3];

let numeri2 = [4, 5, 6];

let tuttiNumeri = numeri1.concat(numeri2);

console.log(tuttiNumeri); // Output: [1, 2, 3, 4, 5, 6]

* **Aggiornamento** di un elemento ad uno specifico indice

let numeri = [1, 2, 3];

numeri[0] = 8; // Aggiorno l’elemento dell’array di indice 0

console.log(numeri); // Output: [8, 2, 3]

### Accesso agli elementi del DOM

Il Document Object Model (DOM) è una rappresentazione ad albero della struttura HTML di una pagina web. Ogni nodo nell'albero rappresenta un elemento HTML, e JavaScript fornisce vari modi per accedere e manipolare questi elementi.

* Accesso **via ID**: ogni elemento HTML al quale è stato specificato un identificativo tramite l’attributo ID è accessibile utilizzando document.getElementById("ID\_ELEMENTO").

Ricordatevi che ciò che viene restituito è **l’oggetto del DOM a cui l’ID viene assegnato** pertanto se si tratta di un elemento <input> voi state accedendo a quell’oggetto e **non direttamente al suo contenuto**.

Dato che gli ID devono essere univoci, la funzione getElementById() restituisce sempre **un** elemento.

let txtQuantitaPrimi = document.getElementById("txtQuantitaPrimi");

* Accesso via **Classname**: come la precedente però non viene effettuata una ricerca via ID ma via nome della classe CSS all’elemento assegnata. A differenza della funzione precedente, ciò che viene restituito è un **array** di elementi visto che una classe CSS può essere condivisa nel DOM.

let elementi = document.getElementsByClassName("classe");

* Accesso via **name**: come la precedente restituisce un array di elementi che hanno lo stesso campo name (per esempio gli elementi dei radio button.

let elementi = document.getElementsByName("name");

### Accesso e modifica alle proprietà degli elementi del DOM

Una volta che l’elemento è stato acquisito in uno dei modi descritti in precedenza, è possibile accedere alle sue proprietà e leggerne o modificarne i valori.

Ricordatevi sempre che le proprietà a disposizione variano in base alla tipologia di elemento al quale state accedendo: per esempio la proprietà checked restituirà un valore solo se l’input è di type="checkbox".

* **Lettura e modifica del testo**: avviene attraverso la proprietà **textContent**. Permette di cambiare o leggere il testo all'interno di un elemento.

lblTotale.textContent = importoNetto.toFixed(2) + " EUR";

* **Lettura e modifica dei valori degli elementi di input**: per accedere o modificare il valore di input negli elementi di un modulo HTML, come campi di testo, checkbox, radio button, select e altri controlli di input viene utilizzata la proprietà **value**. La proprietà value restituisce generalmente una stringa di testo pertanto bisognerà effettuarne la conversione a seconda dell’utilizzo

// Lettura e conversione

let costoBaseAbbonamento = Number(txtCostoAbbonamento.value);

// Assegnazione

txtCostoAbbonamento.value = 1000;

// Lettura e conversione su elementi di tipo Radio Button con value impostato su HTML

for (let i = 0; i < radioElements.length; i++) {

if (radioElements[i].checked) {

      return Number(radioElements[i].value);

      }

}

* **Aggiunta e rimozione di classi CSS**: avviene attraverso l’aggiunta e la rimozione all’array classList. Per esempio, il codice sottostante aggiunge o rimuove una classe chiamata "error" se il valore di una textBox è negativo o positivo

const errorClass = "error";

let value = Number(textBox.value);

if (value < 0) {

    textBox.classList.add(errorClass);

} else {

    textBox.classList.remove(errorClass);

}

* **Modifica** della struttura del DOM: permette di modificare il codice HTML della pagina in maniera tale da aggiungere o rimuovere componenti che verranno visualizzati dal browser

let tbody = tblPersone.getElementsByTagName("tbody")[0];

// Itero su un array persone

for (let i = 0; i < persone.length; i++) {

let persona = persone[i];

// Aggiungo una nuova riga nella tabella

let newRow = tbody.insertRow();

// Creo tre celle nella tabella

let codiceFiscaleCell = newRow.insertCell(0);

let nomeCell = newRow.insertCell(1);

let cognomeCell = newRow.insertCell(2);

// Add some text to the new cells:

codiceFiscaleCell.textContent = persona.codice\_fiscale;

nomeCell.textContent = persona.nome;

cognomeCell.textContent = persona.cognome;

}

* **Aggiunta di eventi on-the-fly** legati a elementi del DOM, per esempio aggiunta in fase di creazione di un onclick su una riga di una tabella creata come descritto in precedenza

// Invoca la funzione per il refresh del form passando come parametro l'oggetto persona relativo

newRow.onclick = function () {

refreshFormPersona(persona.codice\_fiscale);

};

// Alternativamente o nelle Classi che manipolano il DOM

newRow.onclick => () {

refreshFormPersona(persona.codice\_fiscale);

};

### Validazioni

Per ogni input element descritto all’interno della sezione Tag <form> è possibile includere del codice Javascript che possa reagire ad una serie di eventi predefiniti. Utilizzando questi strumenti messi a disposizione di default, è possibile effettuare delle **validazioni grafiche** che sfruttano i principi di modifica on-the-fly del DOM che permettono all’utente finale di ottenere un feedback **visuale** nel caso in cui il dato inserito non sia corretto.

È importante sottolineare come queste validazioni sui dati siano **prettamente grafiche**: per le validazioni logiche delle proprietà di un oggetto, si rimanda alla sezione Classi.

Le validazioni predisposte non fanno altro che **aggiungere** una classe CSS *.error* al componente di input in determinate condizioni (in base alla natura del dato che l’input field deve acquisire) e **rimuoverla** qualora l’informazione inserita sia appartenente al dominio.

In questa fase è possibile una condivisione di codice tra le validazioni logiche e visuali.

**Le validazioni predefinite sono disponibili nel file *formutils.js*.**

### Classi

Una **classe** in JavaScript è un modello per creare oggetti. Definisce le proprietà e i comportamenti (metodi) che gli oggetti creati dalla classe possiederanno. In altre parole, una classe è come uno "schema" da cui possono essere creati diversi oggetti con caratteristiche e comportamenti simili.

Le classi sono utilizzate per organizzare e riutilizzare il codice, facilitando la creazione di oggetti complessi e mantenendo il codice più leggibile e mantenibile.

Le classi permettono di:

* **Raggruppare proprietà e metodi:** Una classe consente di raggruppare attributi (**proprietà**) e funzioni (**metodi**) che sono logicamente correlate in un'unica struttura.
* **Creare più oggetti simili:** Con una classe, puoi creare molteplici oggetti con le stesse proprietà e metodi, ma con dati specifici per ogni istanza.
* **Favorire il riutilizzo del codice:** Una volta definita una classe, puoi riutilizzarla in diverse parti del programma senza dover riscrivere il codice.

Quando, all’interno di una classe, ci si riferisce a parametri o metodi interni, essi vanno **sempre** prefissati con la parola chiave this

class Azienda {

    // Dichiarazione degli attributi privati

    #partitaIva;

    #nome;

    // Costruttore per inizializzare gli attributi

    constructor(partitaIva, nome) {

        this.#partitaIva = partitaIva;

        this.#nome = nome;

    }

    // Getter per l'attributo 'partitaIva'

    get partitaIva() {

        return this.#partitaIva;

    }

    // Setter per l'attributo 'partitaIva'

    set partitaIva(nuovaPartitaIva) {

        this.#partitaIva = nuovaPartitaIva;

    }

    // Getter per l'attributo 'nome'

    get nome() {

        return this.#nome;

    }

    // Setter per l'attributo 'nome'

    set nome(nuovoNome) {

        this.#nome = nuovoNome;

    }

    // Metodo generico

    descrizione() {

        return `Azienda: ${this.nome}, Partita IVA: ${this.partitaIva}`;

    }

}

Così come abbiamo avuto modo di vedere in C#, anche in JavaScript si può sfruttare il meccanismo di **eccezioni** in fase di **validazione di proprietà**.

L’estratto di codice sottostante mostra come, attraverso una **funzione di validazione**, sia possibile gestire l’assegnamento di un valore ad un attributo o il sollevamento di una eccezione attraverso un controllo di validità.

set codice\_fiscale(nuovoCodiceFiscale) {

    if (this.validaCodiceFiscale(nuovoCodiceFiscale)) {

        this.#codice\_fiscale = nuovoCodiceFiscale;

    } else throw new Error("Il codice fiscale non è corretto");

}

/\*\*

  \* Valida il codice fiscale.

  \* Qui possono essere inclusi check molto complessi ma noi per ora ci limitiamo ad una

\* validazione sul numero di caratteri che deve essere 16

  \*/

validaCodiceFiscale(codice\_fiscale) {

    return !this.isNullOrEmpty(codice\_fiscale) && codice\_fiscale.length == 16;

}

Definita la classe, le istanze vengono definite attraverso l’utilizzo della parola chiave new che invocano il codice inserito nel constructor dell’oggetto e ne utilizzano il valore dei parametri

// Creazione di oggetti azienda

let azienda1 = new Azienda("12345678901", "Tech Solutions");

let azienda2 = new Azienda("98765432109", "Innovative Tech");

console.log(azienda1.nome); // Output: Azienda: Tech Solutions, Partita IVA: 12345678901

console.log(azienda2.nome); // Output: Azienda: Innovative Tech, Partita IVA: 98765432109

### Repository (Item Model)

Il Repository Pattern è un Design Pattern che ha lo scopo di astrarre e centralizzare l’accesso ai dati in un'applicazione software. Esso fornisce una **interfaccia uniforme e coerente** per la gestione delle operazioni di persistenza e di recupero dei dati.

In termini pratici, un repository rappresenta una collezione (una **lista**, per esempio) di **oggetti di uno specifico dominio** (Persona, Cliente, Prodotto, ecc), attraverso il quale è possibile es**eguire operazioni comuni come la creazione, la lettura, l'aggiornamento e l'eliminazione di entità** (CRUD).

Internamente e informalmente, abbiamo sempre chiamato le classi che seguono le logiche del Repository Design Pattern come ***classe plurale*** mentre gli oggetti del dominio ***classe singolare***.

**Un Item Model è una implementazione concreta del repository**.

Di seguito un estratto di Item Model basato sulla classe Persona.

class Persone {

    // Variabile privata

    #persone;

    // Costruttore per la classe Persona, le variabili private sono create implicitamente se prefissate con this

    constructor() {

        this.#persone = [];

    }

    // creiamo solo il get così non possiamo riscrivere per intero l'array di Persone

    get persone() {

        return this.#persone;

    }

    // Aggiunge una persona, restituisce true se è stata aggiunta, false altrimenti

    add(persona) {

        if (persona != null && persona instanceof Persona) {

            if (!this.has(persona)) {

                this.persone.push(persona);

                return true;

            }

        }

        return false;

    }

    // Crea una persona. Anche la creazione dell'oggetto passa sempre dal relativo item model

    create(codice\_fiscale, nome, cognome) {

        try {

            let persona = new Persona(codice\_fiscale, nome, cognome);

            return persona;

        } catch (e) {

            throw new Error("Errore nella creazione della persona");

        }

    }

### Fetch API

La Fetch API viene utilizzata per effettuare una richiesta **HTTP**. Restituisce una *Promise* che si risolve con un oggetto *Response*, il quale rappresenta la risposta alla richiesta. Questo oggetto può poi essere elaborato per ottenere i dati richiesti, ad esempio in formato JSON. **La Fetch API permette quindi di usufruire di risorse e servizi raggiungibili via URL**.

La Fetch API è molto potente e versatile, ideale per:

* **Caricare** dati da un server per aggiornare dinamicamente una pagina web.
* **Inviare** dati a un server (ad esempio, dati di un modulo).
* Effettuare **chiamate API** per recuperare o inviare dati a servizi esterni.

**Attenzione!**

La Fetch API necessita di una funzione o metodo di tipo async. Ciò vuol dire che questa parte di codice viene eseguita **parallelamente** al flusso restante del programma (**asincrona**) quindi dobbiamo gestire l’utilizzo dei dati fetchati **solo quando essi sono disponibili**.

Supponiamo di avere un file JSON locale chiamato *aziende.json* che contiene informazioni per creare oggetti Azienda. Ecco un esempio di come utilizzare la Fetch API per scaricare questo JSON e creare oggetti Azienda.

**aziende.json**

[

  {

    "partitaIva": "12345678901",

    "nome": "Tech Solutions"

  },

  {

    "partitaIva": "98765432109",

    "nome": "Innovative Tech"

  }

]

// Funzione asincrona per scaricare il JSON locale e creare oggetti Azienda

async function caricaAziende() {

    try {

        // Usa fetch per recuperare il file JSON

        const response = await fetch('./aziende.json');

        // Controlla se la risposta è OK

        if (!response.ok) {

            throw new Error('Errore nel recupero del file JSON');

        }

        // Converte la risposta in JSON

        const data = await response.json();

        // Ciclo for classico per iterare sugli oggetti nel JSON

        for (let i = 0; i < data.length; i++) {

            let aziendaData = data[i];

            let azienda = new Azienda(aziendaData.partitaIva, aziendaData.nome);

        }

    } catch (error) {

        console.error('Si è verificato un errore:', error);

    }

}

// Chiama la funzione per caricare le aziende

caricaAziende();

Il metodo http utilizzato di default nella fetch API è il **GET**. È possibile utilizzare i restanti metodi http quali **POST, PUT, PATCH, DELETE** configurando opportunamente l’API, opzionalmente definendo il body per i metodi http che lo supportano.

// Upload asincrono della risorsa

const response = await fetch(encodeURI(this.#REMOTEURL + "/" + personaID), {

    method: "PUT", // Metodo HTTP

    headers: {

    "Content-Type": "application/json", // Specifica che invii JSON

    },

    body: JSON.stringify(persona), // Converte l'oggetto in una stringa JSON

});

* **POST**: viene utilizzata per l’inserimento di nuove risorse, accetta opzionalmente un body
* **PUT/PATCH**: vengono utilizzate per l’aggiornamento di risorse, accetta opzionalmente un body
* **DELETE**: viene utilizzata per inviare comandi di eliminazione di risorse

Una seconda modalità per inviare dati tra pagine web o ad un server consiste nell’impostare la **Query String** direttamente nell’URL della chiamata

### Custom Events

I Custom Events (Eventi Personalizzati) in sono una funzionalità che permette di **creare e gestire eventi personalizzati nel tuo codice**. Questo è utile quando hai bisogno di far **comunicare diverse parti della tua applicazione** senza utilizzare direttamente i meccanismi di evento predefiniti: **per esempio notificare l’interfaccia che la Fetch API ha finito di processare una risorsa, visto che essa viene eseguita in maniera asincrona rispetto al flusso del programma**.

Supponiamo di avere una classe Aziende che contenga un array di oggetti di tipo Azienda. Le informazioni di ogni azienda sono su file JSON e scaricate utilizzando la funzione vista in precedenza, che viene estesa includendo il Custom Event a download terminato.

class Aziende {

    // Dichiarazione degli attributi privati

    #listaAziende;

    #downloadedEvt; // Variabile usata per il Custom Event

    // Costruttore per inizializzare gli attributi

    constructor() {

        this.#listaAziende = [];

        this.#fetchAllAziendeEvt = new CustomEvent("aziende-fetched");

    }

    // Getter per l'attributo 'listaAziende'

    get listaAziende () {

        return this.#listaAziende;

    }

// Restituisce il numero di elementi

    size() {

        return this.listaAziende.length;

    }

// Restituisce l’elemento i-esimo

    get(index) {

        if (Number.isInteger(index) && index < this.size()) {

            return this.listaAziende[index];

        }

        return null;

    }

  // Funzione asincrona per scaricare il JSON locale e creare oggetti Azienda

async function fetchAll() {

    try {

        // Usa fetch per recuperare il file JSON

        const response = await fetch('aziende.json');

        // Controlla se la risposta è OK

        if (!response.ok) {

            throw new Error('Errore nel recupero del file JSON');

        }

        // Converte la risposta in JSON

        const data = await response.json();

        // Ciclo for classico per iterare sugli oggetti nel JSON

        for (let i = 0; i < data.length; i++) {

            let aziendaData = data[i];

            let azienda = new Azienda(aziendaData.partitaIva, aziendaData.nome);

            this.listaAziende.push(azienda);

        }

// Se sono stati inseriti elementi invio l’evento

if (listaAziende.length > 0) {

            document.dispatchEvent(this.#fetchAllAziendeEvt);

        }

    } catch (error) {

        console.error('Si è verificato un errore:', error);

    }

}

Gli eventi una volta inviati devono essere **intercettati**. Con il termine intercettati si intendono uno o più componenti che, **autonomamente**, sono in **attesa di ricevere l’evento** che stanno attendendo. L’evento viene propagato su tutti i *listeners* (gli elementi in ascolto) che agiscono in maniera **indipendente**.

Con Javascript possiamo non solo intercettare l’evento ma dire anche **cosa fare** ogni volta che esso viene acquisito.

Supponiamo di avere un file JavaScript (**main.js**) contenente le funzioni che vengono direttamente chiamate da **index.html** tramite gli eventi di default onClick, onChange, onLoad.

**index.html**

<html>

    <head>

        <meta charset="UTF-8" />

        <link rel="stylesheet" href="../css/common.css" />

        <script src="./js/classes/azienda.js"></script>

        <script src="./js/classes/aziende.js"></script>

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

</head>

<body class="centred" onload="init()">

[…]

</body>

L’elemento body invoca la funzione init() che è memorizzata nel file **main.js**.

**main.js**

// Whole-script strict mode syntax

"use strict";

let AZIENDE\_ITEM\_MODEL = new Aziende();

/\*\*

 \* Richiamata subito dopo il caricamento della pagina dall'evento onload del body

 \*/

function init() {

    // Aggiungi un listener per l'evento personalizzato

    document.addEventListener("aziende-fetched", function (event) {

// Quando l’evento viene intercettato richiama la funzione refreshTableAziende()

        refreshTableAziende();

    });

// Faccio partire il download del file JSON delle aziende

    AZIENDE\_ITEM\_MODEL.fetchAll();

}

/\*\*

 \* Effettua il refresh della tabella delle aziende

 \*/

function refreshTableAziende() {

    let tblAziende = document.getElementById("tblAziende");

    let tbody = tblAziende.getElementsByTagName("tbody")[0];

    // Cancello il tbody da tutti i suoi figli così faccio il refresh completo della tabella solo se ci sono dati

    tbody.innerHTML = "";

    for (let i = 0; i < AZIENDE\_ITEM\_MODEL.size(); i++) {

        let azienda = AZIENDE\_ITEM\_MODEL.get(i);

        let newRow = tbody.insertRow();

        let partitaIvaCell = newRow.insertCell(0);

        let nomeCell = newRow.insertCell(1);

        // Aggiungo i valori di ogni oggetto

        codiceFiscaleCell.textContent = azienda.partitaIva;

        nomeCell.textContent = azienda.nome;

    }

}

### Promise

Le *Promises* in JavaScript sono un meccanismo per gestire operazioni asincrone. Una promise rappresenta un valore che potrebbe essere disponibile ora, in futuro, o mai.

Una Promise è **sempre**, anche se non definito, l’output di una funzione che è stata definita come async. Nel nostro caso d’uso, ogni metodo che farà utilizzo della **Fetch Api** che abbiamo visto nella sezione precedente restituirà una Promise.

// Faccio partire il download del file JSON delle aziende

Promise p = AZIENDE\_ITEM\_MODEL.fetchAll();

Ogni istanza di una Promise ha tre metodi principali:

1. .**then**((success) => {}): Esegue un'azione quando la promise è risolta. Il parametro **success** è l’**eventuale** risultato di ritorno della funzione async.
2. .**catch**((error) => {}): Gestisce eventuali errori (quando la promise è respinta). Il parametro **error** è l’**eventuale** errore restituito in una situazione di **try-catch** all’interno della funzione async.
3. .**finally**(): Esegue un'azione a prescindere dall'esito (successo o errore).

L’utilizzo di questi tre metodi potrebbe offrire una alternativa all’utilizzo dei Custom Event per l’aggiornamento dei componenti UI all’arrivo del risultato di una chiamata async; tuttavia, l’utilizzo degli eventi garantisce una migliore **separazione dei ruoli** tra i componenti del proprio programma e offre una maggiore scalabilità in caso di applicazioni complesse.

Tornando alle Promise, oltre che i tre metodi principali offerti da ciascuna istanza offre quelli che sono chiamati **Composition Tools**, ossia degli strumenti che permettono di eseguire parti di codice solo dopo che un numero arbitrario di Promises, **indipendenti una dall’altra**, hanno restituito il loro valore. Ogni metodo restituisce a sua volta una istanza di Promise che quindi può utilizzare i metodi then(), fetch() e finally()

I Metodi di composizione offerti staticamente dalla classe Promise sono i seguenti e ciascuno di essi prende in input l’array di Promises delle quali deve attendere il risultato

1. Promise.**all**([…]): Attende che tutte le promesse siano risolte o si rigetta alla prima promessa rigettata.
2. Promise.**allSettled**([…]): Attende che tutte le promesse siano completate (risolte o rigettate), restituendo i loro stati.
3. Promise.**any**([…]): Restituisce il risultato della prima promessa risolta, rigettandosi solo se tutte vengono rigettate.
4. Promise.**race**([…]): Restituisce il risultato della prima promessa risolta o rigettata.

// Faccio partire il download del file JSON di **due** risorse e **dopo** che entrambe sono state risolte correttamente faccio partire il download di una **terza** risorsa

Promise.all([

CITTA\_ITEM\_MODEL.fetchAll(), // Restituisce una Promise

STATI\_ORDINE\_ITEM\_MODEL.fetchAll() // Restituisce una Promise

])

.then((success) => STATI\_ORDINE\_ITEM\_MODEL.fetch(ORDINE\_ID));

.catch((error) => console.log(error));

### Web API Manager

Per semplificare la gestione di invio e ricezione di richieste HTTP verso un servizio di esposizione e manipolazione dati è stata creata la classe di supporto WebApiManager la quale centralizza quanto visto nella sezione **Fetch API**, demandando a ciascun **Repository (Item Model)** solo la gestione della risposta già *castata* e invio di **Custom Events** all’interfaccia.

**ATTENZIONE:** il casting implicito che viene effettuato da WebApiManager prevede che gli attributi degli oggetti JSON ricevuti e le proprietà della classe di oggetti target siano definiti esattamente nello stesso modo, **case-sensitive**.

### Popup

Immagine che contiene testo, schermata, Carattere, design

Descrizione generata automaticamente

### Progressive Web App

TODO

### Web Components

TODO