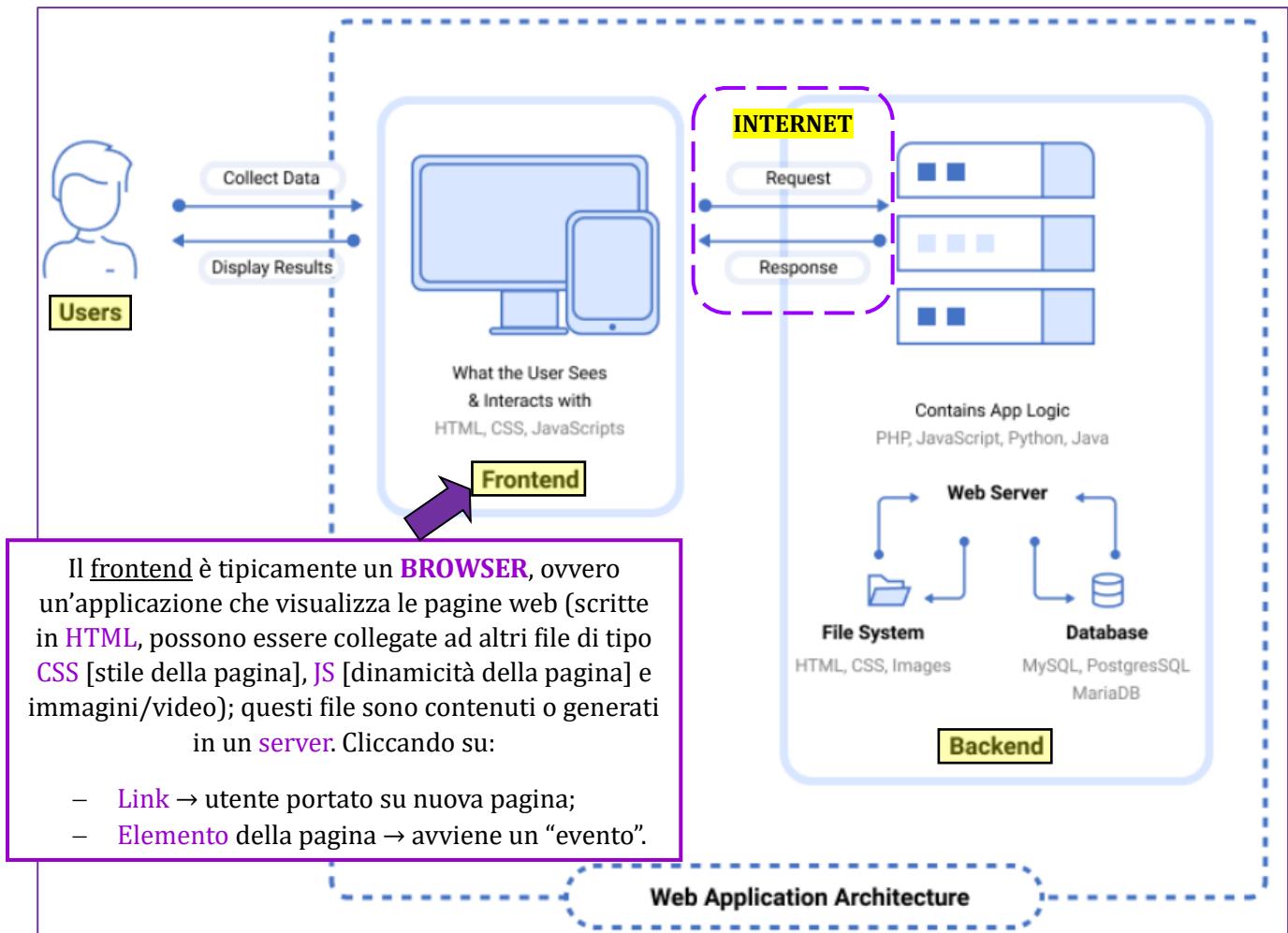


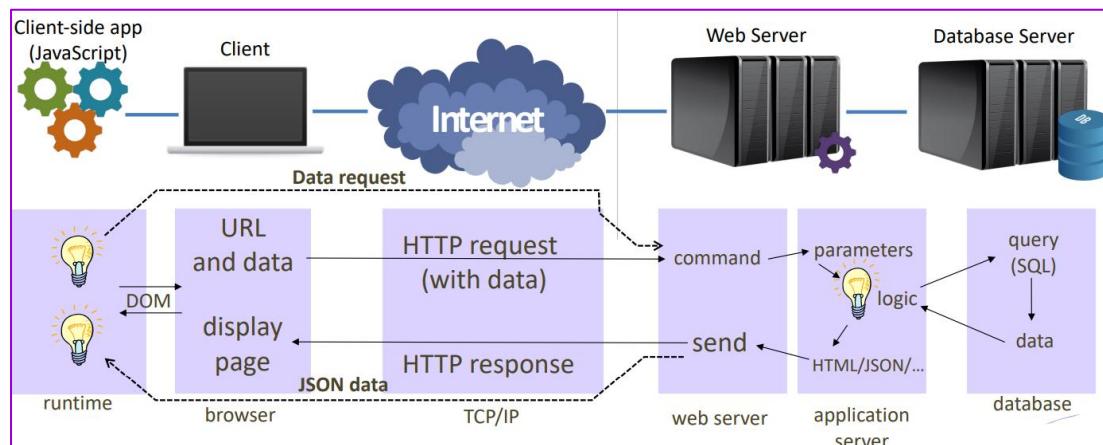
INTRODUZIONE ALLE APPLICAZIONI WEB

1) Architetture Web



La struttura standard delle pagine web è detta **DOM (Document Object Model)** e gestisce gli elementi HTML; i programmi JavaScript possono leggere e modificare questa struttura. Invece i programmi CSS si basano su un insieme di dichiarazioni che modificano alcune proprietà (colori, font, etc...) della pagina operando su alcuni “selettori”, ovvero porzioni del DOM HTML.

La **comunicazione tra Frontend e Backend** avviene attraverso Internet con il protocollo **HTTP** (che prevede 1 richiesta dal **browser** [**frontend**] e 1 risposta dal **server** [**backend**] mediante l'applicazione web che ci gira sopra). Inoltre, sempre nel Backend, vediamo che il server comunica con il “**File System**” e il “**Database**”, dove si trovano le risorse che il server usa per fornire la risposta alla richiesta del browser.



Dallo schema vediamo che, se vogliamo invece aggiornare solo una porzione della pagina, ci può essere del codice JS che “bypassa” tutta la procedura e restituisce un file “JSON” e non HTML.

Un pattern “moderno” è invece “SPA” (SINGLE-PAGE Application), dove il server risponde sempre con la stessa pagina HTML per ogni richiesta che arriva e il lavoro di diversificare il contenuto della pagina lo fa il browser (dunque molto codice JavaScript) [dunque il server ci fornisce i dati, mentre il browser costruisce la pagina].

2) HTML

Useremo uno standard vivo (ovvero che si evolve) moderno (HTML 5, basato non sulla singola pagina, ma sulle applicazioni web [interazione con l’utente e non solo informativi verso l’utente]). Il linguaggio HTML è basato su MARK-UP (tag, attributi etc...). Ma com’è fatto un documento HTML5? Dato che un file html è un albero di elementi innestati uno dentro l’altro (**ELEMENTO = FIGLIO, CONTENITORE = PADRE**):

```
<!DOCTYPE html>      # Intestazione per i documenti html
<html lang="it">    # lang = attributo del tag <html> (lingua della pagina) a cui associo il valore "it"
  <head>
    <title>Pagina Esempio</title>
  </head>
  <body>
    <h1>Pagina Esempio</h1>
    <p>Questo è un paragrafo.</p>
    <!-- commento -->
  </body>
</html>                # Chiudo il file come lo avevo aperto
```

Nell’**head** metto il titolo della pagina
Nel **body** metto il corpo della pagina
<h1> = **titolo principale** (il più importante)
<p> = **paragrafo**
I **commenti** si fanno così!

⚠ In vecchi siti le lettere accentate (in questo caso “è”) si scrivevano con “**&egrave**” oppure “**&acute**”!

⚠ Tramite la modalità “esamina” dei browser, possiamo vedere il codice sorgente e gli elementi del file html; possiamo inoltre visualizzare quanto la pagina è responsive (visualizzare il sito con diverse proporzioni) e vedere le modifiche in tempo reale se vogliamo testarle (cambiare colori etc...) [+ validatori html].

⚠ Il contenuto di **<head>** non sarà visualizzato nel browser, ma serve per la “gestione” della pagina (se devo introdurre dei file o per il titolo della pagina).

Gli **elementi HTML** si dividono in:

- tipo **BLOCK** (**display:block**) → ogni elemento genera una nuova linea e tende ad occupare tutto lo spazio possibile in orizzontale (dall’alto in basso) [es. titoli e paragrafi → **<h1>**, **<p>**];
- tipo **INLINE** (**display:inline**) → ogni elemento non genera una nuova linea e occupa solo spazio necessario (da sinistra a destra), cioè si posizionano uno affianco all’altro e, a fine pagina, vanno a capo [es. link → **<a>**].

```
<a href="#prova">Link</a> # Esempio di link: <a> = ancora (🔗) e il suo attributo "href" (dice dove il
                           <p id="prova">Testo</p> link deve andare); con "#" è un link interno alla pagina stessa, mentre senza
                           è un link ad altre pagine (<a href="pagina2.html">Link</a> oppure <a
                           href="https://polito.it">Link</a>)
```

⚠ L’attributo “**id**” appartiene ad ogni tag ed identifica univocamente un elemento html (se 2 id hanno lo stesso nome, il programma ci porta al 1º)!

Possiamo dividere gli elementi HTML a seconda della loro funzione:

- **SEZIONE** (identificano le sezioni della pagina):
 - **footer** = fondo della pagina;
 - **header** = intestazione (all’inizio del body);

- **section** = sezione generica;
 - **nav** = navbar (navigazione della pagina);
 - **aside** = sezione laterale;
 - **article** = contenuto principale.
- **INTESTAZIONE** (titoli delle sezioni):
- **h1 – h6** = header principali (in ordine di importanza, cresce la dimensione del testo);
 - **hgroup** = raggruppa vari header.
- **GROUPING** (elementi di tipo **block**):
- **p** = paragrafo;
 - Elenchi (liste) contenenti **** (list element):
 - **ol** = elenco ordinato (1,2,3,4);
 - **ul** = elenco senz'ordine (elenco puntato);
 - **dl** = elenco di definizioni (es. vocabolario);
 - **menu** = menu;
 - **main** = testo centrale del documento;
 - **div** = non ha significato semantico, dunque è un contenitore generico per altri tag.
- **PHRASING** (testo contenuto **inline**):
- **a** = ancora per i link;
 - **img** = inserire immagini;
 - **span** = come il div, ma inline;
 - **strong** (o **b**) = parola in grassetto;
 - **em** (o **i**) = parola in corsivo (italico);
 - **button** = bottone su cui cliccare;
 - **canvas** = drag e drop;
 - **video** = video;
 - **input** = campo in cui inserire informazioni;
 - **select** = menu a tendina;
 - **textarea** = campo di testo in cui chi visita il sito scrive.
- **FLOW** (contengono gli altri elementi).

⚠ Le **tabelle** invece sono definite da **<table>** e contengono elementi come **<thead>** (intestazione tabella), **<tbody>** (corpo tabella), **<tfoot>** (footer tabella); ogni riga è definita da **<tr>** e può contenere **<td>** (dati) e **<th>** (intestazione). Le tabelle sono poco responsive in HTML!

⚠ Oltre a “**id**” (che identifica un attributo), possiamo applicare a tutti gli elementi anche l’attributo “**class**” (che contiene più attributi) e “**style**” (che ci permette di applicare delle dichiarazioni CSS ad un file HTML)!

ESERCIZIO – HTML

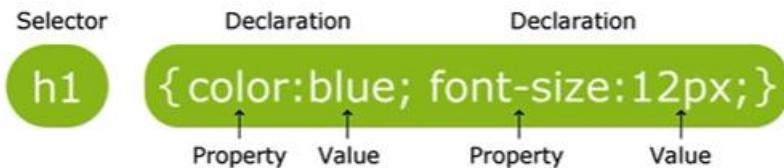
```
<!-- Realizzeremo un BLOG dove avremo:
--- HEADER ---
- titolo del blog + sottotitolo (per l'argomento del blog)
- navigazione (home, contatti, presentazione, login [con opzione per iscriversi, per i commenti...])
--- ASIDE ---
- i 2 articoli più visualizzati del mese + i 2 migliori commentatori (+ ricerca)
--- MAIN ---
- i 3 tag più popolari (a mo' di macro-categoria)
  - lista di 5 articoli (post) ordinati per data (da più nuovo a più vecchio); ognuno con titolo e data (obbligatori), tag (almeno 1), excerpt, immagine (obbligatorio), link che manda alla versione intera dell'articolo e se ci sono dei commenti (quanti)
--- FOOTER ---
- copyright (con anno di creazione) + icone social e email (dell'autore)
-->
```

```
<!DOCTYPE html>
<html lang = "it">
    <head>
        <meta charset = "utf-8">
        <title>IAW - Homepage</title>
        <meta name = "keywords" content = "web, blog, html, css">
        <!-- Qui inserisco lo stylesheet (quando faremo CSS) -->
        <link rel="stylesheet" type="text/css" href="style.css">
    </head>
    <body>
        <header>
            <div>
                <h1>IAW Blog</h1>
                <h2>Tutto su Introduzione alle Applicazioni Web</h2>
            </div>
            <nav>
                <ul>
                    <li><a href="home.html" title = "Homepage">Home</a></li>
                    <li><a href="presenta.html" title = "Di cosa parlo">Presentazione</a></li>
                    <li><a href="contatti.html" title = "Chi sono">Contatti</a></li>
                </ul>
                <a href="login.html" title = "Entra nel sito">Login</a>
            </nav>
        </header>
        <main>
            <article>
                <h3><a href="css.html" title="Post sul CSS">CSS</a></h3>
                <p>2022-10-10</p>
                <p><strong>Tag:</strong> <a href="#">css</a></p>
                <p>CSS sta per Cascading Style Sheet e in questo post ne parleremo meglio.</p>
                <!-- Devo settare bene le dimensioni dell'immagine -->
                
                <p><a href="css.html" title="Commenti al post sul CSS">1 commento</a></p>
            </article>
        </main>
        <aside>
            <div>
                <h3>I più visti del momento</h3>
                <ul>
                    <li><a href="1.html" title="Primo post">Primo titolo</a></li>
                    <li><a href="2.html" title="Secondo post">Secondo titolo</a></li>
                    <li><a href="3.html" title="Terzo post">Terzo titolo</a></li>
                    <li><a href="4.html" title="Quarto post">Quarto titolo</a></li>
                </ul>
            </div>
            <div>
                <h3>I migliori commentatori</h3>
                <ul>
                    <li><a href="p1.html" title="Primo utente">Gianni</a></li>
                    <li><a href="p2.html" title="Secondo utente">Francesco</a></li>
                    <li><a href="p3.html" title="Terzo utente">Roberta</a></li>
                </ul>
            </div>
        </aside>
        <footer>
            <p>2022-2023 IAW - Politecnico di Torino</p>
            <p>Instagram</p><p><a href="mailto:test@polito.it">email</a></p>
        </footer>
    </body>
</html>
```

⚠ Sebbene noi useremo molto più spesso <div> al posto dei tanti caratteri semantici (<nav>, ...), l'obiettivo in questo corso è creare applicazioni web accessibili da tutti (anche chi ha problemi visivi); dunque usare la semantica e inserire "title =..." può essere utile per l'**accessibilità** (soprattutto durante l'ispezione della pagina)!

3) CSS

Il linguaggio **CSS** (Cascading Style Sheets, in questo caso CSS3) è basato su delle **REGOLE** con struttura:



- **SELETTORE** → seleziona a quali tag del file HTML posso applicare la regola (può essere semplice come quello in figura o più complesso se inseriamo delle altre condizioni). I selettori possono essere raggruppati se inserisco più selettori (separati da una virgola) davanti a un blocco di dichiarazioni;
- **DICHIARAZIONE** → un'espressione composta dalla **PROPRIETÀ** (caratteristica che vado a modificare) e dal **VALORE** (associato a quella proprietà).

⚠ Se ci sono 2 regole che vanno in conflitto, 1 che tocca il body in generale (e quindi ogni suo elemento [**EREDITARIETÀ**]) e 1 che tocca un elemento contenuto in body più specifico, la regola che deve seguire l'elemento nel body è quella più specifica!

Ci sono più di 200 proprietà (a cui posso associare un set di valori) che si classificano in base alla loro funzione (animazioni, sfondo, bordi, colori, dimensioni etc...). Queste proprietà seguono determinate **UNITÀ DI MISURA** (assolute, come il pixel [px], e relative); si tende ad usare le unità relative, in quanto quelle assolute potrebbero non adattarsi (per esempio, non sappiamo la densità di pixel dello schermo), vediamone alcune:

- **em** = ci dice che la dimensione del font dell'elemento è X volte quella del contenitore che lo contiene;
- **rem** = ci dice che la dimensione del font dell'elemento è X volte quello dell'elemento <html>;
- **%** = dimensione dell'elemento è % della dimensione del contenitore;
- **vw** (viewport width) = equivale all'1% della larghezza della viewport (ovvero la finestra del browser);
- **vh** (viewport height) = equivale all'1% dell'altezza della viewport (ovvero la finestra del browser).

Ci sono vari tipi di **SELETTORI**, a seconda di cosa seleziono [nelle parentesi quadre, vediamo come scriverli]:

- **ELEMENT** selector → seleziono gli elementi con un certo tag [**element{...}**]);
- **CLASS** selector → seleziono gli elementi con una certa classe [**.class{...}**]);
- **ID** selector → seleziono gli elementi con un certo id (nome) [**#id{...}**]);
- **ATTRIBUTE** selector → seleziono tutti gli elementi con un certo "attributo = ..." [**attribute = val{...}**]).

Posso anche interporre un simbolo davanti all'uguale per cambiare la selezione:

- **attribute** → seleziono tutti gli elementi con quell'attributo;
- **attribute = val** → seleziono tutti gli elementi con quell'attributo = val;
- **attribute ~= val** → seleziono tutti gli elementi con attributo = qualcosa che contiene "val";
- **attribute |= val** → seleziono tutti gli elementi con attributo = qualcosa che inizia per "val";
- **a[attribute ^= val]** → seleziono gli elementi <a> con attributo = qualcosa che inizia per "val";
- **a[attribute \$= val]** → seleziono gli elementi <a> con attributo = qualcosa che finisce per "val"
- **a[attribute *= val]** → seleziono gli elementi <a> con attributo = qualcosa che contiene "val".

- **PSEUDO** selector → seleziono genericamente [:**something{...}**]; esempio:
 - **a:link{...}** → seleziono i link;
 - **a:visited{...}** → seleziono i link visitati;
 - **a:hover{...}** → seleziono il link su cui si trova il mouse in questo momento;
 - **a:active{...}** → seleziono i link attivi attualmente;
 - **tr:hover{...}** → seleziono la casella su cui si trova il mouse in questo momento;
 - **input:focus{...}** → seleziono il campo di testo dove ho cliccato attualmente.

⚠ Questi selettori si possono combinare tra loro:

- **S1, S2** → S1 unito a S2

- **S1 S2** → S2 inserito in S1
- **S1 > S2** → S2 figlio di S1
- **S1 + S2** → S2 viene subito dopo S1
- **S1 ~ S2** → S2 viene preceduto da S1

⚠ Abbiamo anche la proprietà **display** (che è quella di cui abbiamo parlato con gli elementi inline e block) che ci permette di cambiare un elemento inline in block e viceversa (**{display:inline;}** o **{display:block;}**). C'è anche **{display:none;}** che ci permette di eliminare un elemento dalla pagina (da non confondere con **{visibility:hidden;}** che invece lo nasconde e basta, ma non lo elimina)!

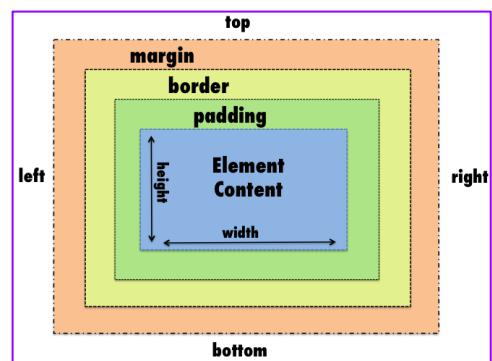
Il termine **CASCADING** contenuto in CSS indica che un documento può includere più style sheets: in questo caso si guarda la gerarchia (**INLINE STYLE** [nel tag `<html>`], **INTERNAL STYLE** [nell'head dell'`html`], **EXTERNAL STYLE**, **BROWSER DEFAULT STYLE**):

- **INLINE STYLE** → lo implemento all'interno dell'elemento singolo (rimane uguale per ogni elemento con lo stesso tag), quindi evitabile;
- **INTERNAL STYLE** → lo implemento sempre in `<head>`, ma non da un file esterno con un link, bensì usando l'elemento `<style type="text/css"> h1{font-size:17px;...} </style>`;
- **EXTERNAL STYLE** → lo implemento attraverso un link del tipo `<link rel=stylesheet type="text/css" href="style.css">` dentro `<head>`;

⚠ Se inseriamo il termine **!important** nello stile, questo avrà priorità maggiore rispetto a quelli senza questo termine → `h1 {color:red !important}`. Inoltre, anche i selettori hanno una loro gerarchia di priorità (ID, classi, attributi, pseudo, elementi).

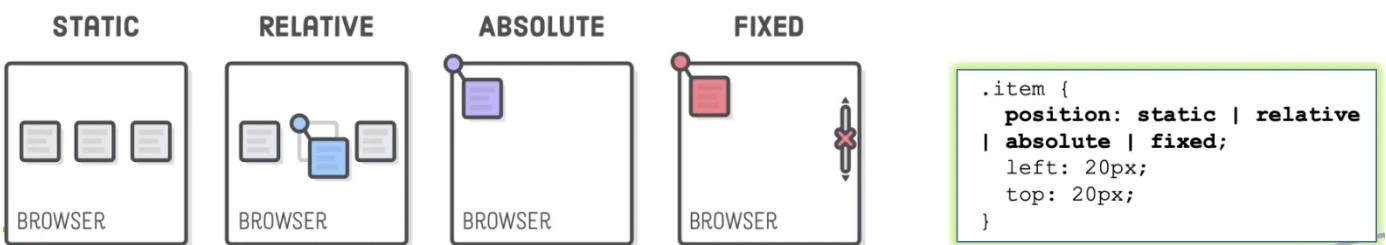
Il modello alla base del CSS è il **BOX MODEL**, ovvero ogni elemento di una pagina HTML è una scatola rettangolare con struttura:

- **CONTENT** (contenuto) = testo/immagine;
- **PADDING** (margine interno) = area intorno al contenuto, legata al colore di sfondo del box;
- **BORDER** (bordo) = esterno al padding, legato anch'esso al colore di sfondo del box;
- **MARGIN** (margine esterno) = area trasparente intorno al bordo.



Per quanto riguarda il **POSIZIONAMENTO** di un elemento nella pagina:

- **STATIC** = default, ovvero elementi inline stanno sulla stessa riga mentre block vanno a capo;
- **RELATIVE** = offset relativo alla posizione del blocco di default;
- **ABSOLUTE** = la posizione del box è determinata dalla posizione del padre (il contenitore) e rimane fissa (ovvero se scrolla la pagina, l'elemento scompare in quanto rimane fisso dov'è);
- **FIXED** = come l'assoluta, ma rimane ancorato dov'è (ovvero se scrolla la pagina, l'elemento rimarrà visibile sempre nella stessa posizione).



⚠ C'è anche la proprietà **z-index:numero** che serve a creare dei layers per visualizzare gli elementi in primo piano, secondo piano etc...

Per quanto riguarda i metodi di **LAYOUT** della pagina abbiamo i seguenti metodi:

- **FLOAT** → proprietà che dà il controllo sulla posizione orizzontale di un elemento (posso usare il **clear** per stoppare);



CONTINUAZIONE ESERCIZIO – CSS FLOAT (BASE)

```
/* File style.css affiancato all'HTML visto in precedenza */

body {
    padding-left: 1rem;
    padding-right: 1rem;
}

.post-img {
    width: 25%;
}

aside {
    float: left;
    width: 30%; /* Devo settare una certa larghezza per vedere il layout giusto */
    margin-top: 8%;
    background-color: aliceblue;
}

main {
    float: right;
    width: 65%;
    margin-top: 8%; /* Per evitare che main e aside finiscano sotto header */
    margin-left: 1rem;
}

footer {
    clear: both; /* Elimino il layout float dal footer */
}

header {
    background-color: lightskyblue;
    position: absolute;
    width: 100%;
}

header > div {
    float: left;
    width: 40%;
}

header > nav {
    float: right;
    width: 40%;
}

article {
    border-color: black;
    border-width: 2px;
    border-style: dashed;
}
```

⚠ Oltre al FLOAT, abbiamo anche [layout più avanzati](#) (anche perché noi implementeremo i layout, non li costruiremo da zero con il codice):

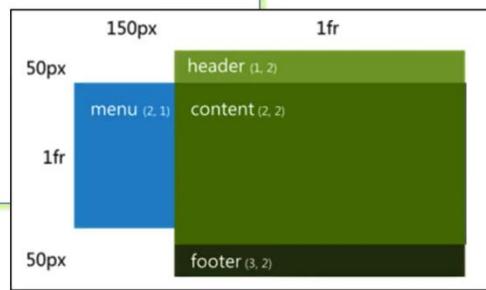
- **GRID → layout a griglia** (non tabella). Si può definire esplicitamente (**grid-columns** e **grid-rows** con proprietà) o automaticamente (CSS gestisce la griglia):

	Maki-zushi The rice and seaweed rolls with fish and/or vegetables. There are also more specific terms for the rolls depending on the style.		Nigiri-zushi A single finger of rice topped with wasabi and a fillet of raw or cooked fish or shrimp. Generally the most common form of sushi you will see.		Temaki-zushi Also called a hand-roll. Consists of sushi rice, fish and vegetables wrapped in seaweed. It is very similar to maki.
	WHAT IS SUSHI? Beginning as a method of preserving fish centuries ago, sushi has evolved into an art, without losing importance in its original form. Dried fish was placed between two pieces of vinegared rice as a way of making it last. The nori (seaweed) was added later as a way to keep one's fingers from getting sticky.		Technically, the word "sushi" refers to the rice, but colloquially, the term is used to describe a finger-size piece of raw fish or seafood on a bed of vinegared rice or simply the consumption of raw fish in the Japanese style (when sushi is not solely a Japanese invention, these days, the Japanese style is considered the de facto serving standard).		MAKI-ZUSHI Beginning as a way of preserving fish centuries ago, as soon as it was invented, there were specific terms for the rolls depending on the style.
	SASHIMI Sashimi is raw fish served sliced, but as is. That means it is not cooked, but it is often served alongside daikon and/or shiso. This is my favorite style as you really get the flavor of the fish.		SUSHI FOR PARTIES: MAKI-ZUSHI AND NIGIRI-ZUSHI This book has great pictures. However it is not as complete as Sushi Made Easy.		NIGIRI-ZUSHI The most popular form of sushi is nigiri-zushi. It consists of a single finger of rice topped with a piece of raw fish and seaweed. Generally the most common form of sushi you will see.
	QUICK & EASY SUSHI COOKBOOK This book has great pictures. However it is not as complete as Sushi Made Easy.		SUSHI FOR PARTIES: MAKI-ZUSHI AND NIGIRI-ZUSHI This book has great pictures, with advanced maki (roll) making techniques.		TEMAKI-ZUSHI Also called a hand-roll. Consists of sushi rice, fish and vegetables wrapped in seaweed. It is very similar to maki.

```

section {
  display: grid;
  grid-template-columns: 150px 1fr;
  grid-template-rows: 50px 1fr 50px;
}

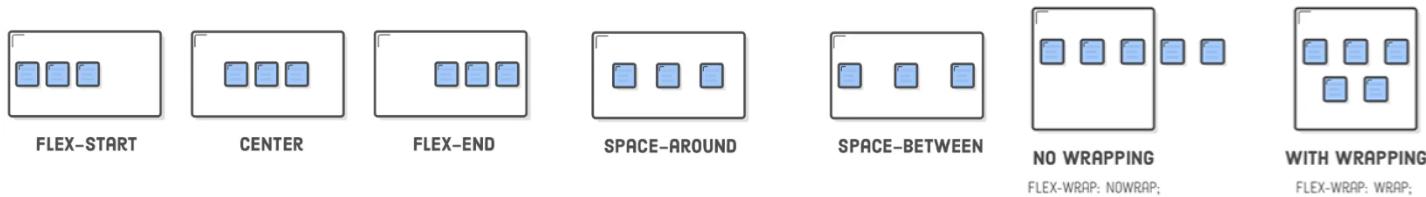
```



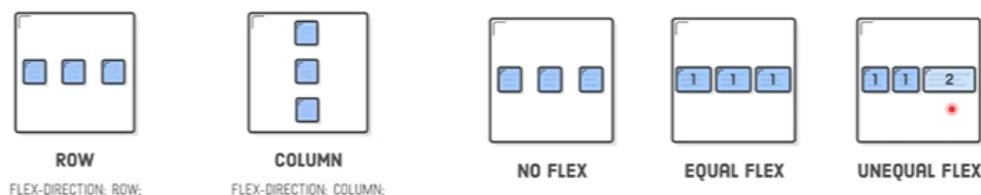
→ Con le griglie si usa l'unità di misura **fr** (valore **frazionario**), ovvero la frazione della pagina che vogliamo occupare (in questo esempio infatti dividiamo lo spazio orizzontale in 150 px e tutto il resto [1 fr]).

→ Per trasformare un elemento HTML in una **grid** si usa { **display: grid;** }!

- **FLEX** → molto più **flessibile** di float e grid, in quanto abbiamo completo controllo (ovvero allineamento, direzione, ordine, dimensione) su ogni box (**flexbox**). Ci sono 2 tipi di box: **flex container** (ovvero il contenitore dei flex items che definisce il loro posizionamento) e **flex item** (singolo elemento). Per trasformare un elemento HTML in un **flex container** si usa { **display: flex;** } ed è utile la proprietà **justify-content** (che definisce l'allineamento orizzontale dei flex items nel container) così come la **flex-wrap** (che ci dice se andare a capo o meno, finito lo spazio) e la **align-items** (come **justify-content** ma verticale):



Sono anche importanti la **flex-direction** (che dice se gli items sono disposti in riga/orizzontali {**flex-direction: row;**} o colonna/verticali {**flex-direction: column;**}) e la **flex** (che dice se gli items si possono stretchare e come):



⚠ I flex-items si possono raggruppare usando dei <div> come contenitori (che saranno quindi a loro volta contenuti nei flex-containers).

Tutti questi layout (float [con inline e block], grid e flex) sono dei valori (**keywords**) della proprietà **display**; **flex**, **grid**, **table** sono **inside** keywords (contesto), mentre **block**, **inline** sono **outside** keywords (ruolo nel flusso) [infatti scrivere {**display: block**} equivale a {**display: block flow**}, mentre {**display: flex**} equivale a {**display: block flex**}]. Un'altra inside keyword è **flow-root** (permette di ridefinire il flusso, contenuto nel tag <html>, soltanto in un altro elemento) [o **inline-block**].

⚠ Per capire la differenza effettiva nei vari layout, andare a vedere i 3 css contenuti in “[cssUpgradeEsempio](#)”!

→ Ultima cosa importante riguardo il CSS è il **RESPONSIVE LAYOUT**, che **non solo scala** la nostra pagina web a seconda del dispositivo usato, **ma la modifica** per adattarla nel modo migliore al dispositivo con cui viene visualizzata. Per costruire un responsive design, si deve racchiudere la regola css in una [**MEDIA QUERY**](#):

```
@media(min-width:900px){p{color:red;}}
```

Media query announcement

What circumstance
should this query be
“turned on” or applied

What it should do if the
circumstance happens

Nelle queries si può aggiungere il carattere **print** (per le stampanti) o **screen** (dispositivi con schermo); inoltre si possono anche combinare e negare più queries mediante gli operatori booleani [es. **@media screen and (min-height: 680px)**, **not (orientation:portrait){...}** → applica lo stile se è uno schermo **e** ha altezza minima di 680px **oppure** se l'orientamento **non** è portrait] (si può usare anche il “**compreso**” a $a \leq x \leq b$).

CONTINUAZIONE – CSS FLEX + RESPONSIVE (nell'HTML `<div class = "container">` raggruppa `<main>` e `<aside>`)

```
.post-img {  
    width: 25%;  
}  
  
header {  
    display: flex;  
    justify-content: space-between;  
    background-color: aliceblue;  
}  
  
header > div {  
    flex: 2;          /* Riduce la distanza tra il titolo e il div e nav (stretcando)*/  
    padding-left: 1rem;  
}  
  
header > nav {  
    flex: 1;          /* Il flex = 1 fa da riferimento per il flex sopra (il doppio) */  
}  
  
.container {  
    display: flex;  
    flex-direction: row-reverse;      /* Per mettere aside a sinistra*/  
}  
  
main {  
    flex: 3;  
}  
  
aside {  
    flex: 1;  
    background-color: rgb(231, 231, 231);    /* Altro modo per definire i colori */  
    /*background-color: #E7E7E7*/  
}  
  
/* Ora cerchiamo di renderlo responsive */  
@media screen and (max-width: 600px) {  
    .container {  
        flex-direction: column; /* alternativa: display: block; */  
    }  
  
    aside {  
        display: none; /* fa scomparire la barra laterale se ristretto */  
    }  
}
```

⚠ Molte pagine web usano una struttura a griglia per essere responsive (la singola cella si adatta allo schermo)!

Nella pratica, non scriveremo mai pagine web da zero, ma useremo un **framework** (per alleggerire il carico di lavoro) chiamato **BOOTSTRAP** (basato sul concetto di classi **container**, che contengono delle classi **righe**, le quali a loro volta contengono delle classi **colonne**) [guarda esempi sulla [documentazione di Bootstrap](#)]; in esso troviamo già fatte una struttura base delle navbar, dei bottoni, caroselli, modali, cards etc... Inoltre, ognuno di questi elementi è reso **responsive** dal JavaScript ad esso associato.

Per usare bootstrap all'interno di un file html, devo importare con un link il CSS a cui fa riferimento (e importare con uno script il JS a cui fa riferimento, ma ne parleremo in seguito di JS).

CONTINUAZIONE - BOOTSTRAP (faremo la pagina web su cui stiamo lavorando in bootstrap)

```
<!DOCTYPE html>
<html lang = "it">
    <head>
        <meta charset = "utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>IAW - Homepage</title>
        <meta name = "keywords" content = "web, blog, html, css">
        <!-- Importo il CSS di bootstrap con il link -->
        <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet"
              integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/DwykC2MPK8M2HN"
              crossorigin="anonymous">
            <!-- Posso anche aggiungere un file css fatto da me, dopo quello di bootstrap -->
            <link href="style.css" rel="stylesheet">
    </head>
    <body>
        <header>
            <!-- Importo la navbar dal sito di bootstrap e la modifico per inserire i nostri elementi -->
            <nav class="navbar fixed-top navbar-dark navbar-expand-lg bg-primary">
                <div class="container-fluid">
                    <!-- Modifico il titolo per renderlo non cliccabile (no link, solo testo)-->
                    <span class="navbar-brand mb-0 h1">IAW Blog</span>
                    <button class="navbar-toggler" type="button"
                           data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent"
                           aria-controls="navbarSupportedContent" aria-expanded="false"
                           aria-label="Toggle navigation">
                        <span class="navbar-toggler-icon"></span>
                    </button>
                    <div class="collapse navbar-collapse" id="navbarSupportedContent">
                        <ul class="navbar-nav me-auto mb-2 mb-lg-0">
                            <li class="nav-item">
                                <a class="nav-link active" aria-current="page" href="#">Home</a>
                            </li>
                            <li class="nav-item">
                                <a class="nav-link" href="#">Presentazione</a>
                            </li>
                            <li class="nav-item">
                                <a class="nav-link" href="#">Contatti</a>
                            </li>
                        </ul>
                    </div>
                </div>
            </nav>
        </header>
        <!-- Ora uso la struttura a griglia su cui è basato bootstrap -->
        <div class="container-fluid under-nav" > <!-- under-nav lo uso per il margine dal nostro style.css
-->
            <div class="row">
                <!-- Se schermo large = 8, altrimenti 12 -->
                <main class="col-lg-8 col-12">
                    <article>
                        <h3><a href="css.html" title="Post sul CSS">CSS</a></h3>
                        <p>2022-10-10</p>
                        <p><strong>Tag:</strong> <a href="#">css</a></p>
                        <p>CSS sta per Cascading Style Sheet e in questo post ne parleremo meglio.</p>
                        
                        <p><a href="css.html#commenti" title="Commenti al post sul CSS">1
commento</a></p>
                    </article>
                </main>
                <!-- Se schermo large = 4, altrimenti 12 -->
                <aside class="col-lg-4 col-12">
                    <div>
                        <h3>I più visti del momento</h3>
                        <ul>
                            <li><a href="1.html" title="Primo post">Primo titolo</a></li>
                            <li><a href="2.html" title="Secondo post">Secondo titolo</a></li>
                            <li><a href="3.html" title="Terzo post">Terzo titolo</a></li>
                            <li><a href="4.html" title="Quarto post">Quarto titolo</a></li>
                        </ul>
                    </div>
                </aside>
            </div>
        </div>
    </body>

```

```

        </div>
        <div>
            <h3>I migliori commentatori</h3>
            <ul>
                <li><a href="p1.html" title="Primo utente">Gianni</a></li>
                <li><a href="p2.html" title="Secondo utente">Francesco</a></li>
                <li><a href="p3.html" title="Terzo utente">Roberta</a></li>
            </ul>
        </div>
    </aside>
</div>
<footer>
    <p>2022-2023 IAW - Politecnico di Torino</p>
    <p>Instagram</p><p><a href="mailto:test@polito.it">email</a></p>
</footer>
<!-- Importo il JavaScript di bootstrap con lo script e lo metto al fondo del body-->
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"
integrity="sha384-C6RzsynM9kWDrMNeT87bh950GNyZPhcTNXj1NW7RuBCsyN/o0jlpcV8Qyq46cDfL"
crossorigin="anonymous"></script>
</body>
</html>

```

⚠ Il file **style.css** contiene invece `.under-nav {margin-top: 60px;}` e `.post-img {width: 25%;}`!

→ OFF-TOPIC: parliamo di **ARCHITETTURA DELL'INFORMAZIONE** (**Visual design**), ovvero il modo in cui viene strutturato un mezzo di informazione (in questo caso i siti web) basandoci sul **contenuto**, sull'**utente** e sul **contesto**. L'obiettivo è rendere il contenuto accessibile e comprensibile (possiamo fare dei **prototipi** per la nostra pagina web [come a benessere digitale], come le "sitemaps"); il nostro cervello anche senza vedere al completo una pagina di google, sa benissimo la sua struttura e sa cosa aspettarsi visivamente (**pattern**).

4) FLASK

Con **FLASK** iniziamo a parlare di **BACKEND** (lo faremo in Python), ovvero di **SERVER** [quindi useremo il frontend che abbiamo imparato, ma per i server (in Python); non lo faremo a mano con l'HTTP ma anche qui useremo un **framework** che si chiama **FLASK**]. Questo micro-framework si basa su 2 tools: **Werkzeug** (server) e **Jinja** (HTML engine editing).

Per lavorare con Flask devo usarlo in un **venv** (virtual environment): dopo aver installato python3, vado nella cartella in cui voglio usare Flask e li (tramite **Powershell**) uso il comando `python3 -m venv venv` per creare la sottocartella dell'ambiente virtuale; a questo punto, una volta che ho installato tutto, uso il comando `venv\Scripts\activate` per lavorare nell'ambiente virtuale (e `deactivate` per smettere di usarlo). A questo punto creo il file python su cui lavorare, importo Flask [`from flask import Flask`] e creo l'applicazione web [`app = Flask(__name__)`].

Quando invece voglio runnare il file creato, devo usare il comando `flask --app nomeFile run` (e se il file si chiama `app.py` posso usare semplicemente `flask run`). In questo modo si crea un **server apposito** (il cui indirizzo http ci viene scritto nel terminale) e ci viene fornito da flask l'**errore 404** (quando utente fa una richiesta ad un indirizzo inesistente); nel terminale troviamo anche il comando per quittare il server e tutto il **log** delle operazioni eseguite. Come ulteriori comandi si possono aggiungere `--debug` (per lanciarlo in debug), `--host=0.0.0.0` (per lanciare un server pubblico) e `--port=3000` (per cambiare la porta, in questo caso mettendola a 3000).

⚠ Per comodità, possiamo usare il classico comando `python` per lanciare l'applicazione (`python3 app.py`) se inseriamo nel codice della nostra app:

```

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=3000, debug= True)
    # or a subset of these options

```

→ Infatti come abbiamo già visto questo è il **punto di ingresso di ogni programma python** quando viene lanciato direttamente (una sorta di funzione `main`) [che appunto non funziona se lo importo!]

⚠ A differenza dei file HTML, per vedere le modifiche effettuate nel codice di un server, dobbiamo quittare e far ripartire il server con il comando `flask run`; proprio per questo si può usare la **debug mode**, che oltre a farci vedere gli errori prende le modifiche live.

Ogni pagina web in flask è rappresentata dalla coppia **DECORATOR + FUNZIONE**:

- **DECORATOR** → `@app.route(path, options)`
 - o `app` = l'oggetto di flask;
 - o `path` = percorso sul server, ovvero il percorso nella richiesta http che il server riceve dal browser;
 - o `options` = opzioni. Le più importanti sono *methods* (ovvero a quali metodi HTTP quella path deve rispondere, di default GET) e *redirect_to* (ovvero fa un redirect ad un altro indirizzo, utile per spostarci tra una pagina e l'altra).
- **FUNZIONE** → `def index(): [a capo] return HTML content`
 - o `index` = nome della pagina;
 - o `HTML content` = contenuto HTML della pagina (es. 'Hello world!');
 - o `request.method` = variabile di default per ogni funzione con il metodo HTTP richiesto dal browser).

Esempio

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return """<html><head><title>IAW- Home</title></head>
    <body><h1>Blog di Introduzione alle Applicazioni web</h1>
    <p>Benvenuto sul blog del corso.</p>
    <p></p>
    <p>&copy; <a href="about.html">Introduzione alle Applicazioni web</a></p>
    </body></html>"""

@app.route('/about.html')
def about():
    return """<html><head><title>IAW - Sul corso</title></head>
    <body><h1>Introduzione alle Applicazioni Web - Informazioni</h1>
    <p>Il corso &egrave; offerto al terzo anno delle lauree in Ingegneria al Politecnico di Torino.</p>
    <p>Questo esempio &egrave; stato creato nell'anno accademico 2022/2023.</p>
    <p><a href="/">Torna al blog</a></p>
    </body></html>"""

# Usare gli URL come stringhe è sconveniente perchè se cambio il path devo cambiare tutto;
# perciò flask ci mette a disposizione url_for('<function_name>') per le funzioni,
# che si può usare anche per i file statici url_for('static', filename='image.jpg')
# [i file statici per convenzione vanno messi nella sottocartella "static"]

from flask import Flask, url_for

app = Flask(__name__)

@app.route('/')
def index():
    return """<html><head><title>IAW- Home</title></head>
    <body><h1>Blog di Introduzione alle Applicazioni web</h1>
    <p>Benvenuto sul blog del corso.</p>
    <p><img src='""" + url_for('static', filename='logo.png') + """'></p>
    <p>&copy; <a href='""" + url_for('about') + """'>Introduzione alle Applicazioni web</a></p>
    </body></html>"""

@app.route('/about.html')
def about():
    return """<html><head><title>IAW - Sul corso</title></head>
    <body><h1>Introduzione alle Applicazioni Web - Informazioni</h1>
    <p>Il corso &egrave; offerto al terzo anno delle lauree in Ingegneria al Politecnico di Torino.</p>
    <p>Questo esempio &egrave; stato creato nell'anno accademico 2022/2023.</p>
    <p><a href="/">Torna al blog</a></p>
    </body></html>"""
```

Il problema che persiste è che serve scrivere **moltissimo codice HTML** nel return per restituirlo al browser; la soluzione ci viene fornita con l'**HTML engine di Flask**, ovvero **Jinja** che tramite il **templating** ci permette di **non scrivere codice HTML in Python**. Per fare ciò dobbiamo definire dei file HTML a parte (i template) nella sottocartella “`/templates`”; il template va inserito nel return con `return render_template('file.html', name=name)`. La dicitura `name=name` ci dice che il 2° nome è il nome della variabile del file app.py (il solito `_name_`), mentre il 1° è il nome con cui verrà chiamata la variabile nel template HTML (solitamente si usa lo stesso nome apposta).

Scrivendo le **variabili**, le **espressioni** e le **funzioni** python con doppia parentesi graffa `{{parametro}}`, queste verranno tradotte in codice HTML dal template; nel template possiamo fare lo stesso con gli **statement** (es. if, cicli etc...) `{%statement%}` e con i **commenti** (non inclusi nell'output finale) `{#commento#}`. Per esempio, posso inserire la funzione `url_for('<function_name>')` oppure `url_for('static', filename='image.jpg')` dentro `{}{...}`, mentre posso inserire dei loop con `{% for variable in list %} ... {% endfor%}` e degli if con `{% if condizione %} ... {% elif condizione %} ... {% else %} ... {% endif %}`.

! Se la variabile python dovesse contenere del testo contenente dei caratteri speciali in HTML, bisogna inserire una **sequenza di escape** al fondo (es. `{}{ variabile |e }}`)!

In Jinja c'è l'**EREDITARIETÀ** tra i template (per non dover scrive ogni volta gli stessi blocchi HTML più volte); si usa `{% block <block_name>%} ... {% endblock %}` per rendere riusabile il **blocco** di codice HTML, e `{% extends "file.html" %}` per rendere riusabile l'intero **template** da altri template. Infatti proprio per non dover ripetere ogni volta la struttura base di un file HTML, si può usare l'ereditarietà per definire un file `base.html` del tipo:

```
<!DOCTYPE html>
<html lang = "it">
    <head>
        <title>IAW - {{ block title }}{% endblock %}</title>
    </head>
    <body>
        {{ block content }}{% endblock %}
    </body>
</html>
```

```
templates/index.html
{% extends "base.html" %}
{% block title %}Blog{% endblock %}
{% block content %}
    <h1>Blog di Introduzione alle
    Applicazioni Web</h1>
    <p>Benvenuto sul blog del corso.</p>
    <p>
        
    </p>
    <p>&copy; <a href="{{ url_for('about') }}>Introduzione alle
    Applicazioni Web</a></p>
{% endblock %}
```

! Con la scrittura `{}{ super() }`, possiamo invece prendere un blocco e aggiungerne delle informazioni ulteriori senza sovrascriverlo!

Legato al templating ci sono anche le **DYNAMIC ROUTES**, dove possiamo usare un **parametro** (passato nelle tonde del decorator) all'interno della funzione; verrà considerato come una **stringa**, quindi se vogliamo che venga

```
@app.route('/users/<username>')
def show_profile(username):
    return 'User %s' % username
```

considerato diversamente (es. un intero) dobbiamo specificarne la **conversione** (es. `<int: id>` [int, float, path (ovvero stringa con /)]).

Infatti, esiste anche la versione di `url_for` con il parametro [es. `url_for('show_profile', username='mario')`]

Flask è un microframework **estendibile**, ovvero possiamo aggiungere delle **estensioni** per:

- **Flask-Mail** → aggiunge la possibilità di mandare mail dal server;
- **Flask-Session** → aggiunge il supporto alle sessioni lato-server;
- **Flask-Login** → gestire le sessioni degli utenti loggati (infatti usa anche Flask-Session);
- **Bootstrap-Flask** → prende dei pezzi di bootstrap e li mette dentro i template HTML;
- **Flask-SQLAlchemy** → integrazione con SQLAlchemy (lavorare coi database senza query SQL, ma con degli oggetti di programmazione);
- **Flask-WTF** → integrazione con WTForms (Form = oggetto con dei campi di input che viene poi inviato [es. la casella di login dove inserisco utente e password]).

CONTINUAZIONE – FLASK (+ tutto quello imparato finora)

→ **app.py**

```
from flask import Flask, render_template

app = Flask(__name__)

# creo la struttura dati per i post
posts = [
    {'id':1,'title':'CSS', 'date':'2022-10-10', 'tag':'css', 'content':'CSS sta per Cascading Style Sheet e in questo post ne parleremo meglio.'},
    {'id':2,'title':'HTML', 'date':'2022-10-04', 'tag':'html', 'content':'HTML sta per HyperText Markup Language e in questo post ne parleremo meglio.'}
]

@app.route('/')
def index():
    return render_template('index.html', posts=posts)

@app.route('/posts/<int:id>')
def single_post(id):
    post = posts[id-1]
    return render_template('single.html', post=post)

@app.route('/about')
def about():
    return render_template('about.html')
```

→ **base.html [struttura base]**

```
<!DOCTYPE html>
<html lang = "it">
    <head>
        <meta charset = "utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>IAW - {% block title %}{% endblock %}</title>
        <meta name = "keywords" content = "web, blog, html, css">
        <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-DfXdzD2hBM021QBDZab9qElPhHbWz/50iSpLkja4D0PqKJ1jy03UPLnJ8R7R" crossorigin="anonymous">
        <link href="{{ url_for('static', filename='style.css') }}" rel="stylesheet">
    </head>
    <body>
        <header>
            <nav class="navbar fixed-top navbar-dark navbar-expand-lg bg-primary">
                <div class="container-fluid">
                    <span class="navbar-brand mb-0 h1">IAW Blog</span>
                    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
                        <span class="navbar-toggler-icon"></span>
                    </button>
                    <div class="collapse navbar-collapse" id="navbarSupportedContent">
                        <ul class="navbar-nav me-auto mb-2 mb-lg-0">
                            <li class="nav-item">
                                <a class="nav-link {% block home_active %} {% endblock %}" aria-current="page" href="/">Home</a>
                            </li>
                            <li class="nav-item">
                                <a class="nav-link {% block about_active %} {% endblock %}" href="{{ url_for('about') }}>Presentazione</a>
                            </li>
                            <li class="nav-item">
                                <a class="nav-link {% block contacts_active %} {% endblock %}" href="#">Contatti</a>
                            </li>
                        </ul>
                    </div>
                </div>
            </nav>
        </header>
        <main>
            <div class="py-4" style="text-align: center;">
                <h1>IAW - Istruzioni di Accesso e Utilizzo</h1>
                <p>Questo è un sito web di esempio. Ecco le informazioni chiave: <b>Nome Utente: IAW</b> e <b>Password: Istruzioni di Accesso e Utilizzo</b>. Per ulteriori dettagli, visitate la <a href="/about">Presentazione</a> o <a href="#">Contatti</a> se avete domande.</p>
            </div>
        </main>
    </body>
</html>
```

```

        </div>
    </div>
</nav>
</header>
<div class="container-fluid under-nav">
    <div class="row">
        <main class="col-lg-8 col-12">
            {% block content %} {% endblock %}
        </main>
        <aside class="col-lg-4 col-12">
            {% block sidebar %} {% endblock %}
        </aside>
    </div>
</div>
<footer>
    <p>2022-2023 IAW - Politecnico di Torino</p>
    <p>Instagram</p><p><a href="mailto:test@polito.it">email</a></p>
</footer>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"
integrity="sha384-C6RzsynM9kWDrMNeT87bh95OGNyZPhcTNxj1NW7RuBCsyN/o0jlpcV8Qyq46cDfL"
crossorigin="anonymous"></script>
</body>
</html>
```

→ **index.html** [Home]

```

{% extends "base.html" %}
{% block title %} Home {% endblock %}
{% block home_active %} active {% endblock %}
{% block content %}
    {%# Ora che ho la lista di posts, ciclo for per stamparla tutta. #}
    {% for post in posts%}
        <article>
            <h3><a href="{{url_for('single_post', id=post.id)}}>{{post.title}}</a></h3>
            <p>{{post.date}}</p>
            <p><strong>Tag:</strong> <a href="#">{{post.tag}}</a></p>
            <p>{{post.content}}</p>
            
            <p><a href="1.html#commenti" title="Commenti al post sul {{post.title}}">1 commento</a></p>
        </article>
    {% endfor %}
    {% endblock %}
    {% block sidebar %}
        <div>
            <h3>I più visti del momento</h3>
            <ul>
                <li><a href="#" title="Primo post">Primo titolo</a></li>
                <li><a href="#" title="Secondo post">Secondo titolo</a></li>
                <li><a href="#" title="Terzo post">Terzo titolo</a></li>
                <li><a href="#" title="Quarto post">Quarto titolo</a></li>
            </ul>
        </div>
        <div>
            <h3>I migliori commentatori</h3>
            <ul>
                <li><a href="#" title="Primo utente">Gianni</a></li>
                <li><a href="#" title="Secondo utente">Francesco</a></li>
                <li><a href="#" title="Terzo utente">Roberta</a></li>
            </ul>
        </div>
    {% endblock %}
```

→ **about.html** [Presentazione]

```

{% extends "base.html" %}
{% block title %} Presentazione {% endblock %}
{% block about_active %} active {% endblock %}
```

```

{% block content %}
    <p> Blog di IAW. </p>
{% endblock %}

{% block sidebar %}
<div>
    <h3>I più visti del momento</h3>
    <ul>
        <li><a href="#" title="Primo post">Primo titolo</a></li>
        <li><a href="#" title="Secondo post">Secondo titolo</a></li>
        <li><a href="#" title="Terzo post">Terzo titolo</a></li>
        <li><a href="#" title="Quarto post">Quarto titolo</a></li>
    </ul>
</div>
<div>
    <h3>I migliori commentatori</h3>
    <ul>
        <li><a href="#" title="Primo utente">Gianni</a></li>
        <li><a href="#" title="Secondo utente">Francesco</a></li>
        <li><a href="#" title="Terzo utente">Roberta</a></li>
    </ul>
</div>
{% endblock %}

```

→ **single.html [Singoli post]**

```

{% extends "base.html" %}
{% block title %}Blog: {{post.title}}{% endblock %}
{% block content %}
<article>
    <h3>{{post.title}}</h3>
    <p>{{post.date}}</p>
    <p><strong>Tag:</strong> <a href="#">{{post.tag}}</a></p>
    <p>{{post.content}}</p>
    
    <h4>Commenti (2)</h4>
    <ol>
        <li>Bel post!</li>
        <li>Vero.</li>
    </ol>
</article>
{% endblock %}

```

⚠ A differenza di python (dove per accedere all'elemento di un dizionario devo usare le quadre con la chiave, ovvero `post["title"]` oppure `post.get("title")`), qui Jinja mi fornisce un metodo alternativo per accedere all'elemento del dizionario, più simile al C e JavaScript, ovvero con il **punto (`post.title`)**.

5) FORMS

Con **Forms**, intendiamo un **pezzo della pagina che gli utenti possono compilare e inviare** (dunque il form è HTML, ma l'invio delle informazioni è lato server [quindi Flask]). Per definire un form in HTML, uso il tag **<form>**, che ha tutti gli attributi già visti negli altri tag, ma ne ha anche 2 specifici, ovvero **action** (indica l'URL a cui le informazioni compilate nel form vanno inviate) e **method** (metodo http con cui si invia un form; se non viene scritto, di default è GET → un form con una GET avrebbe poi l'URL contenente le informazioni inserite[es. se `action = "http://bla-bla"` e nel form inserisco `username` e `password`, avrò che l'URL finale sarà `http://bla-bla/login?username=gigio&password=1234`]). Dunque è sconsigliato usare GET come method perché le informazioni inserite dall'utente vengono visualizzate (si preferisce il method POST o PUT). A differenza degli altri elementi HTML, che mi consentono di vedere informazioni, i **controlli dei form** mi permettono di inserire informazioni; vediamone alcuni:

- **<input>** → tag per inserire elementi; per esempio, **<input type="text" name="firstname" placeholder="Your firstname">** ci dice che dobbiamo inserire del testo, che il testo ha **name** "firstname" e che nella casella di

testo ci verrà mostrato "Your firstname" prima di inserire il testo (**placeholder**). Il **type** può essere **button**, **checkbox**, **color**, **date**, **email**, **file**, **hidden**, **month**, **number**, **password**, **radio**, **range**, **submit**, **reset**, **search**, **tel**, **text**, **url**, **week** (per esempio password farà comparire i pallini quando digito, mentre uso checkbox per le selezioni multiple e radio per le selezioni singole). Abbiamo anche **attributi** come **required**, **readonly**, **checked** etc... che verificano condizioni **generali** su questi input (es. required = campo obbligatorio), ma anche attributi **specifici** che si applicano solo al tipo di input (es. max e min per il tipo number, oppure accept per il tipo file per specificare il tipo di file accettato);

- **<label>** → tag che introduce l'input, in quanto è un'etichetta **legata ad un certo input**; per esempio, l'etichetta `<label for="mela"> Ti piace la mela? </label>` si lega all'input `<input type="checkbox" name="mela" id="mela">` grazie all'associazione tra il campo **for** (per la label) e **id** (per l'input). È importante per motivi di accessibilità;
- **<select>** → tag per "selezione a tendina". Mettiamo al suo interno i tag **<option>** (ovvero i singoli elementi da selezionare nello scorrimento), che però possiamo anche raggruppare ulteriormente con **<optgroup>** `label= "nome_gruppo">`;
- **<button>** → oltre all'input **type button**, esiste proprio il tag **button** che prevede 3 **type**: **submit** (manda il form al server), **reset** (resetta il form), **button** (comportamento da definire in JavaScript). Il button, rispetto all'input type button, è meglio, in quanto è **più flessibile** e può contenere immagini.

⚠ Dato che la visualizzazione del form dipende dal browser in uso, si consiglia di usare dei framework/librerie piuttosto che i singoli elementi del form da zero!

Dato che il form invia delle informazioni ad un server, oltre alla **client-side validation** (ovvero controllare che le stringhe inserite siano corrette, che i campi numerati rispettino eventuali condizioni etc... [**required**, **minlength**, **maxlength**, **min**, **max**, **type**, **pattern**] → possiamo definire nel css **input:invalid** e **input:valid**, in modo che, se i campi inseriti rispettano/non rispettano le regole, accada visivamente qualcosa al riquadro di input), ci va anche la **server-side validation** (per esempio, se nel campo nome un utente inserisce un comando python che cancella dei dati, devo agire per evitare che questo accada) [DOPPIA VALIDAZIONE]. Per esempio, in un login, una volta superata con successo la client-side validation, il server verifica se lo username esiste e se la password ad esso associata è corretta.

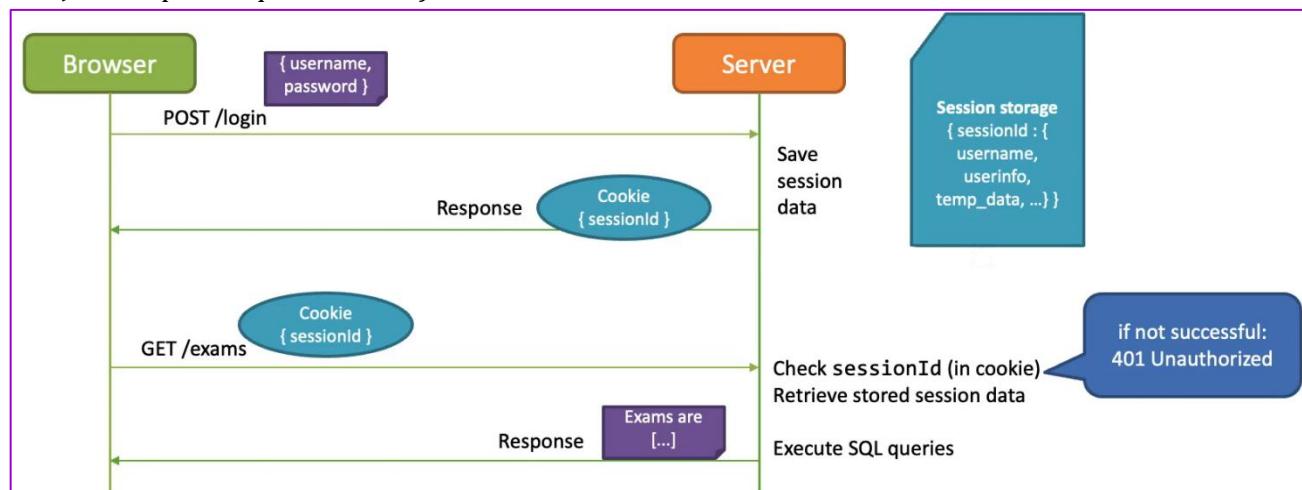
Dunque l'intero contenuto del form compilato viene inviato al server con una richiesta HTTP (POST o PUT); se invece uso **Flask**, l'intero contenuto verrà impacchettato nella variabile **request** (che rappresenta il pacchetto HTTP in python che il server riceve e dunque **request.form** rappresenta il campi del form). Il metodo **request** va importato da flask e il **request.form** è un **dizionario**, dunque per accedere ad un campo singolo del form posso usare **request.form['nome']** oppure **request.form.get('nome')** [più sicuro per keyerror].

Le **validazioni server-side** possono essere fatte manualmente con gli **if** oppure usando estensioni di Flask come WTForm. Le informazioni ottenute dal form (dopo la validazione) le useremo anche nei return render_template [es. se studente o docente, potrebbero caricarsi 2 pagine diverse con **return render_template('studente.html', ruolo=ruolo_dal_form)** oppure **return render_template('docente.html', ruolo=ruolo_dal_form)**; inoltre se volessi usare poi la variabile **ruolo** la posso usare normalmente nel mio HTML con le doppie graffe `{} ruolo {}`]. Si usano anche **app.logger.debug('Testo da visualizzare per il debug')** [oltre a **debug**, **warning** e **error**] per **stampare del testo a schermo** in situazioni di debug, warning e error.

Se in un form inseriamo dei campi **file**, questo va specificato con l'attributo **enctype='multipart/form-data'** e per accedere/salvare questi file devo prenderli con **request.files['nome_file']**, salvarli in una variabile e poi usare questa variabile con il **.save('percorso_di_salvataggio/nuovo_nome_file')** [se vogliamo recuperare il nome originale del file, possiamo usare **secure_filename(variabile_dove_salvato_file.filename)**].

⚠ Con **request.form**, i valori dei campi spariscono subito (temporanei); dunque, potremmo salvarli in un **database** (lo vedremo più avanti) o memorizzare alcuni di questi campi in una **sessione**: ricordandoci che ogni richiesta HTTP è indipendente dalle altre (non si possono passare le informazioni tra le pagine), una soluzione a questo problema sono le sессии (un modo di memorizzare e scambiare informazioni temporaneamente). Queste sessioni usano i **cookies** (piccole informazioni memorizzate nel cookie storage del browser, inviate dal browser al server [dunque non devono contenere informazioni sensibili]); un cookie ha attributi **name**, **value**, **expiration**

date, secure (invia dal browser solo se usiamo HTTPS e non HTTP), httpOnly (non può essere toccato da codice JavaScript, dunque sicurezza).



Flask prevede le sessioni, ma solo lato client, ovvero memorizzo in cookies la struttura dati **session** con i campi che mi servono (es. `session['user'] = name` che posso riusare con `{} session['user'] {}`); data la minore sicurezza del lato client, si tende comunque a preferire la session lato server (noi infatti non useremo Flask, ma useremo una sua estensione [Flask-Sessions] che implementa le sessioni lato server). La sessione si chiude se chiudo il browser o se elimino i file di sessione nella cartella, altrimenti anche refreshando la pagina, questa rimane con la sessione attiva.

CONTINUAZIONE – FORMS (scrivo solo le modifiche effettuate rispetto alla continuazione precedente)

→ `app.py`

```

from flask import Flask, render_template, request, redirect, url_for, flash, session
from datetime import date
from flask_session import Session

app = Flask(__name__)
# La riga sotto posso toglierla perchè il flashing viene effettuato con la session
#app.secret_key = 'questa è una secret key'

# Queste 2 istruzioni mi fanno passare da client-side session (cookies)
# a server-side session; con config, configuro dove salvo la sessione
app.config['SESSION_TYPE'] = 'filesystem'
app.config['SESSION_PERMANENT'] = False      # sessione non permanente
Session(app)

# creo la struttura dati per i post
posts = [
    {'id':1,'title':'CSS', 'date':'2022-10-10', 'tag':'css',
     'content':'CSS sta per Cascading Style Sheet e in questo post ne parleremo meglio.'},
    {'id':2,'title':'HTML', 'date':'2022-10-04', 'tag':'html',
     'content':'HTML sta per HyperText Markup Language e in questo post ne parleremo meglio.'}
]

@app.route('/')
def index():
    return render_template('index.html', posts=posts)

@app.route('/posts/<int:id>')
def single_post(id):
    post = posts[id-1]
    return render_template('single.html', post=post)

@app.route('/about')
def about():
    return render_template('about.html')

@app.route('/posts/new', methods=['GET', 'POST'])

```

```

def new_post():
    if request.method == 'POST':
        post = request.form.to_dict() # rende request.form modificabile
        # gestire errore titolo
        if post['title'] == '':
            app.logger.error('Titolo non può essere vuoto!')
        # data default
        if post['date'] == '':
            post['date'] = date.today()
        # salvo immagine dal form se esiste (per questo aggiungo l'enctype)
        post_image = request.files['image']
        if post_image:
            post_image.save('static/' + post['title'] + '.jpg')
        #aggiungo id (metto in coda il nuovo post)
        post['id'] = posts[-1]['id'] + 1
        #aggiungo post all'array posts
        posts.append(post)
        # per visualizzare messaggio di successo
        flash('!!!Post creato correttamente!!!', 'success')
        return redirect(url_for('index'))
    else:
        return render_template('new-post.html')

@app.route('/amministratore')
def admin():
    # se viene premuto il bottone amministratore
    session['admin'] = True
    return redirect(url_for('index'))

→base.html

<!DOCTYPE html>
<html lang = "it">
    <head>
        <meta charset = "utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>IAW - {% block title %}{% endblock %}</title>
        <meta name = "keywords" content = "web, blog, html, css">
        <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/DwykC2MPK8M2HN" crossorigin="anonymous">
        <link href="{{ url_for('static', filename='style.css') }}" rel="stylesheet">
    </head>
    <body>
        <header>
            <nav class="navbar fixed-top navbar-dark navbar-expand-lg bg-primary">
                <div class="container-fluid">
                    <span class="navbar-brand mb-0 h1">IAW Blog</span>
                    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
                        <span class="navbar-toggler-icon"></span>
                    </button>
                    <div class="collapse navbar-collapse" id="navbarSupportedContent">
                        <ul class="navbar-nav me-auto mb-2 mb-lg-0">
                            <li class="nav-item">
                                <a class="nav-link {% block home_active %} {% endblock %}" aria-current="page" href="/">Home</a>
                            </li>
                            <li class="nav-item">
                                <a class="nav-link {% block about_active %} {% endblock %}" href="{{ url_for('about') }}>Presentazione</a>
                            </li>
                            <li class="nav-item">
                                <a class="nav-link {% block contacts_active %} {% endblock %}" href="#">Contatti</a>
                            </li>
                        </ul>
                    </div>
                </div>
            </nav>
        </header>
        <div class="container">
            <div class="row">
                <div class="col-12 col-md-8 mb-4">

```

```

        {% if session.admin %}
            <a href="{{ url_for('new_post') }}" class="btn btn-outline-light"
                title="Crea nuovo post">Nuovo post</a>
        {% else %}
            <!-- aggiungo un bottone di admin prima di creare nuovo post -->
            <a href="{{ url_for('admin') }}" class="btn btn-outline-light"
                title="Diventa amministratore"> Amministratore </a>
        {% endif %}
    </div>
</div>
</header>
<div class="container-fluid under-nav">
    <div class="row">
        <!-- Aggiungo il contenitore per i messaggi flash con alert di bootstrap -->
        {% with messages = get_flashed_messages(with_categories=true) %}
            {% if messages %}
                {% for category, message in messages %}
                    <div class="alert alert-{{category}} alert-dismissible fade show"
                        role="alert">
                        {{ message }}
                        <button type="button" class="btn-close" data-bs-dismiss="alert"
                            aria-label="Close"></button>
                    </div>
                {% endfor %}
            {% endif %}
        {% endwith %}
    <main class="col-lg-8 col-12">
        {% block content %} {% endblock %}
    </main>
    <aside class="col-lg-4 col-12">
        {% block sidebar %} {% endblock %}
    </aside>
</div>
</div>
<footer>
    <p>2022-2023 IAW - Politecnico di Torino</p>
    <p>Instagram</p><p><a href="mailto:test@polito.it">email</a></p>
</footer>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"
integrity="sha384-C6RzsynM9kWDrMNeT87bh95OGNyZPhcTNXj1NW7RuBCsyN/o0jlpcV8Qyq46cDFL"
crossorigin="anonymous"></script>
</body>
</html>

```

→ new-post.html

```

{% extends "base.html" %}
{% block title %}New post{% endblock %}
{% block content %}
    <!-- Prendo un esempio di form dalla documentazione di bootstrap -->
    <form action="/posts/new" method="POST" enctype="multipart/form-data">
        <div class="mb-3">
            <label for="title" class="form-label">Titolo</label>
            <input type="text" class="form-control" name="title" aria-describedby="Titolo del post"
                required minlength="3" maxlength="30">
        </div>
        <div class="mb-3">
            <label for="date" class="form-label">Data</label>
            <input type="date" class="form-control" name="date" aria-describedby="Data post">
        </div>
        <div class="mb-3">
            <label for="tag" class="form-label">Tag</label>
            <input type="text" class="form-control" name="tag" aria-describedby="Tag del post">
        </div>
        <div class="mb-3">
            <label for="content" class="form-label">Contenuto</label>
            <textarea class="form-control" name="content" rows="3" required
                minlength="10"></textarea>
        </div>
    </form>

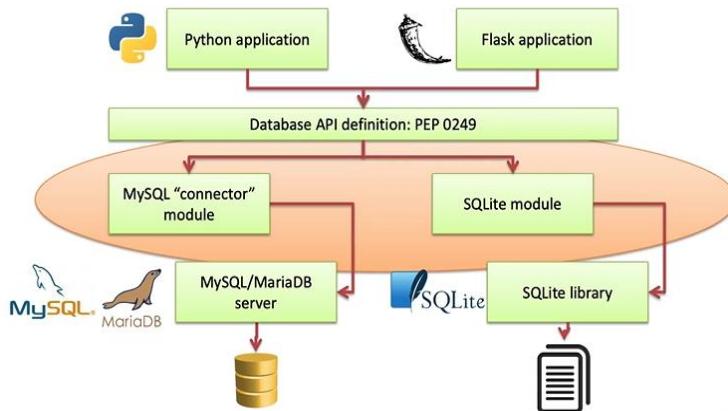
```

```

</div>
<div class="mb-3">
    <label for="image" class="form-label">Immagine</label>
    <input type="file" class="form-control" name="image" aria-describedby="Carica img">
</div>
<button type="submit" class="btn btn-primary">Crea</button>
</form>
{% endblock %}

```

6) DATABASE (in Python)



Con un'applicazione python e un DBMS [server] (es. MySQL), vogliamo interagire con il database; noi però non vogliamo scaricare un altro server (perché vogliamo fare cose facili in questo corso), ma useremo SQLite (ovvero un “database su file”). Python aggiunge un’interfaccia (un PEP, ovvero un Python Enhancement Proposal) per la connessione tra l’applicazione python e il database, che rende i nostri programmi indipendenti dal database usato (basterà cambiare solo 1 istruzione per accedere ad altri database, il resto rimane uguale).

⚠ Bisogna sempre usare placeholder [es. “`INSERT INTO translation (original, modified) VALUES (?, ?)`”] invece della concatenazione di stringhe [es. “`INSERT INTO translation (original, modified) VALUES (“ + stringa1 + “, ”+ stringa2 “)`”] per motivi di sicurezza.

Quali sono gli step per interagire con un database:

1. Scrivere le queries SQL memorizzandole nella variabile sql come una stringa (usando i placeholder con ?) [es. `sql = “SELECT id, original, modified FROM translation”;`]
2. Ci connettiamo al database (SQLite) → `conn = sqlite3.connect('exable.db');`
3. Otteniamo il cursore e eseguiamo la query (se ci sono delle variabili usate nei placeholders dobbiamo inserirle nella tupla in execute):
 - `cursor = conn.cursor()`
 - `cursor.execute(sql)` [o con dei placeholders `cursor.execute(sql, (variabile1, variabile2))` e se ho solo 1 placeholder scriverò `(variabile1,)`]
4. Prendiamo i risultati della query:
 - se ho una SELECT → `cursor.fetchone()` [prendo il prossimo risultato] oppure `cursor.fetchall()` [prendo tutti i risultati rimanenti];
 - se ho un INSERT, UPDATE o DELETE → `conn.commit()` [fa il commit sulle operazioni (successo)];
5. Chiusura: `cursor.close()` [chiudo il cursore] + `conn.close()` [chiude la connessione].

⚠ SQLite3 è già contenuto in python dunque dovrò solo importarlo (`import sqlite3`) e stabilire la connessione (punto 2)! Per fare più operazioni nella stessa connessione, posso definire più cursori in essa.

CONTINUAZIONE – DATABASE (solo modifiche effettuate)

→ `posts_dao.py` [creato per fare le funzioni che mi prendono i dati dal database `blog.db`]

```

# dao sta per data access object
import sqlite3

def get_posts():
    conn = sqlite3.connect('db/blog.db')
    # uso Row che è una sorta di dizionario (per non usare le tuple)
    conn.row_factory = sqlite3.Row

```

```

cursor = conn.cursor()
# li ordino secondo aggiunta più recente
sql = 'SELECT * FROM posts ORDER BY date DESC'
cursor.execute(sql)
posts = cursor.fetchall()
cursor.close()
conn.close()
return posts

def get_post(id):
    conn = sqlite3.connect('db/blog.db')
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()
    sql = 'SELECT * FROM posts WHERE id = ?'
    cursor.execute(sql, (id,))
    post = cursor.fetchone()
    cursor.close()
    conn.close()
    return post

def add(post):
    conn = sqlite3.connect('db/blog.db')
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()
    success = False
    sql = 'INSERT INTO posts(title, date, tag, content, user_id) VALUES(?, ?, ?, ?, ?)'
    # prova a creare il post, se riesce commit, altrimenti rollback (come a basi di dati)
    try:
        cursor.execute(sql, (post['title'], post['date'], post['tag'], post['content'], 1))
        conn.commit()
        success = True
    except Exception as e:
        print('ERROR', str(e))
        conn.rollback()
    cursor.close()
    conn.close()
    return success

def get_comments(id):
    conn = sqlite3.connect('db/blog.db')
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()
    sql = 'SELECT content FROM comments WHERE post_id = ?'
    cursor.execute(sql, (id,))
    comments = cursor.fetchall()
    cursor.close()
    conn.close()
    return comments

```

→ app.py

```

from flask import Flask, render_template, request, redirect, url_for, flash, session
from datetime import date
from flask_session import Session
# importo il file che ho creato per gestire il database (come da teoria)
import posts_dao

app = Flask(__name__)
app.config['SESSION_TYPE'] = 'filesystem'
app.config['SESSION_PERMANENT'] = False
Session(app)

# Sono passato dal dizionario posts al database
@app.route('/')
def index():
    posts = posts_dao.get_posts()
    return render_template('index.html', posts=posts)

```

```

@app.route('/posts/<int:id>')
def single_post(id):
    post = posts_dao.get_post(id)
    comments = posts_dao.get_comments(id)
    return render_template('single.html', post=post, comments=comments)

@app.route('/about')
def about():
    return render_template('about.html')

@app.route('/posts/new', methods=['GET', 'POST'])
def new_post():
    if request.method == 'POST':
        post = request.form.to_dict()
        if post['title'] == '':
            app.logger.error('Titolo non può essere vuoto!')
        if post['date'] == '':
            post['date'] = date.today()
        post_image = request.files['image']
        if post_image:
            post_image.save('static/' + post['title'] + '.jpg')
    # uso la variabile success restituita dalla add per diversificare gli alert
    success = posts_dao.add(post)
    if success:
        flash('!!!Post creato correttamente!!!', 'success')
    else:
        flash('!!!Errore nella creazione del post!!!', 'danger')
    return redirect(url_for('index'))
else:
    return render_template('new-post.html')

@app.route('/amministratore')
def admin():
    session['admin'] = True
    return redirect(url_for('index'))

```

→ **index.html**

Modificando i commenti da posts_dao, devo modificare anche il link a cui fanno riferimento (riga 13):

```
<p><a href="{{url_for('single_post', id=post.id)}}>Vedi commenti</a></p>
```

→ **single.html**: analogamente, modifco anche qui i commenti:

```

{% extends "base.html" %}
{% block title %}Blog: {{post.title}}{% endblock %}
{% block content %}
    <article>
        <h3>{{post.title}}</h3>
        <p>{{post.date}}</p>
        <p><strong>Tag:</strong> <a href="#">{{post.tag}}</a></p>
        <p>{{post.content}}</p>
        
        {% if comments | length > 0 %}
            <h4>Commenti</h4>
            <ol>
                {% for comment in comments %}
                    <li>{{comment.content}}</li>
                {% endfor %}
            </ol>
        {% endif %}
    </article>
{% endblock %}

```

7) AUTENTICAZIONE (Flask)

Bisogna diversificare 2 concetti tra loro legati, ovvero **AUTENTICAZIONE** (verificare l'identità dell'utente con delle credenziali) e **AUTORIZZAZIONE** (il permesso di fare determinate cose, che dipende dall'autenticazione). L'autenticazione include sia il client che il sever e ci sono degli **strati** per essa:

Who	What	How	When
User	Login / Logout / Navigate pages		
Flask App	Is the user logged? Remember user information	Flask-Login	Set at login Destroyed at logout Queried during navigation
Browser	Remembers navigation session	Session Cookie (stores session ID)	Received at login, in HTTP Response Re-sent to server at every HTTP Request
Server	Remember session data	Session storage (creates session ID, remembers associated data: username, group, level, ...)	Created at login Destroyed at logout Retrieved at every HTTP Request
Route (HTTP API)	Check authorization Execute API	Verify session validity	At every (non-public) HTTP Request
Route (Login)	Perform authentication	Check user/pass If ok, create session information	At Login time
Route (Logout)	Forget authentication	Destroy session information	At Logout request
Database (at Login)	Validates user information	Queries & password encryption	At Login time
Database (HTTP API)	Retrieves user information	Queries from session information	At every HTTP Request

Abbiamo già parlato di **cookies** (mantengono piccole quantità di informazioni, lato-client) e **sessioni** (analoghi ma con Flask, lato-server); ora parliamo anche di **session ID** (ovvero dopo che mi autentico, rimango **loggato durante la sessione tramite un id**, fornitomi dal server), che mi serve per fare le chiamate al server.

Ma come funziona nella pratica il login? L'utente **compila il form nel client** (browser) con il suo id e password; questi dati vengono validati (**client-side validation**) e, se è tutto ok, vengono mandati al server, il quale controlla se l'utente è già registrato e se la password corrisponde (**server-side validation**). Se non corrispondono invia messaggio di errore, mentre se sono corretti genera il **session ID** e lo memorizza (con alcune informazioni sull'utente prese dal database) nel “**server session storage**”. Poi il server risponde alla richiesta HTTP di login creando e inviando un **cookie** al browser, il quale lo memorizza e invia la risposta all'applicazione web.

Noi per fare ciò useremo **Flask-Login**, che facilita tutto questo percorso in maniera sicura e rispettando gli standard (installandolo con **npm install flask-login**). Per l'implementazione, dobbiamo però ricordarci di creare un oggetto **user** (**user model**) per indicare cosa intendiamo noi con “utente” (le caratteristiche, gli attributi, le proprietà etc...) [con **UserMixin** vengono implementati i metodi che ci si aspetta di default dall'utente]; mentre con **user_loader** indichiamo dove andare a prendere l'utente. La classe **user** avrà dei metodi (**is_authenticated**, **is_active**, **is_anonymous**, **get_id()** etc...). Sotto la route possiamo aggiungere **@login_required** se è richiesto il login e possiamo usare funzioni come **login_user()** e **logout_user()** [inoltre possiamo usare **current_user** per le informazioni sull'utente loggato]. Per motivi di sicurezza, non possiamo salvare le password così come sono state scritte nel database; infatti useremo delle **funzioni di hash** per alterare le password da salvare [**generate_password_hash(password, method='sha256')**] importata da **werkzeug.security**, che installiamo con **pip install werkzeug**.

```
from flask import Flask
from flask_login import LoginManager

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret'

login_manager = LoginManager()
login_manager.init_app(app)
```

ESEMPIO – LOGIN

→ **app.py** [struttura dell'app con le funzioni di route di autenticazione]

```
from flask import Flask, render_template, request, redirect, url_for, flash
from flask_login import LoginManager, login_user, login_required, logout_user, current_user
from werkzeug.security import generate_password_hash, check_password_hash
import dao
from models import User
```

```

app = Flask(__name__)
app.config['SECRET_KEY'] = '90LWxND4o83j4K4iuop0'
login_manager = LoginManager()
login_manager.init_app(app)

@app.route('/')
def home():
    return render_template('login.html')

@app.route('/iscriviti')
def signup():
    return render_template('signup.html')

@app.route('/iscriviti', methods=['POST'])
def signup_post():
    name = request.form.get('name')
    surname = request.form.get('surname')
    email = request.form.get('email')
    password = request.form.get('password')

    user_in_db = dao.get_user_by_email(email)
    if user_in_db:
        flash('Già un utente registrato con questa email', 'danger')
        return redirect(url_for('signup'))
    else:
        new_user = {"name":name, "surname":surname, "email":email,
                   "password":generate_password_hash(password, method='sha256')}
        success = dao.add_user(new_user)
        if success:
            flash('Utente creato correttamente', 'success')
            return redirect(url_for('login'))
        else:
            flash('Errore nella creazione dell\'utente', 'danger')
            return redirect(url_for('signup'))

@app.route('/accedi')
def login():
    return render_template('login.html')

@app.route('/accedi', methods=['POST'])
def login_post():
    email = request.form.get('email')
    password = request.form.get('password')

    user_in_db = dao.get_user_by_email(email)
    if not user_in_db or not check_password_hash(user_in_db['password'], password):
        flash('Credenziali non valide, riprova', 'danger')
        return redirect(url_for('login'))
    else:
        new_user = User(id=user_in_db['id'], name=user_in_db['name'], surname=user_in_db['surname'],
                        email=user_in_db['email'], password=user_in_db['password'])
        login_user(new_user)
        return redirect(url_for('profile'))

@app.route('/profilo')
@login_required
def profile():
    return render_template('profile.html')

@app.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for('home'))

@login_manager.user_loader
def load_user(user_id):
    user_in_db = dao.get_user_by_id(user_id)
    user = User(id=user_in_db['id'], name=user_in_db['name'], surname=user_in_db['surname'],
                email=user_in_db['email'], password=user_in_db['password'])

```

```

        email=user_in_db['email'], password=user_in_db['password'])
    return user

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=3000, debug=True)

→ dao.py [per le funzioni con cui accedere agli elementi del database]

import sqlite3

def get_user_by_email(email):
    conn = sqlite3.connect('db/users.db')
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()

    sql = 'SELECT * FROM users WHERE email = ?'
    cursor.execute(sql, (email,))
    user = cursor.fetchone()

    cursor.close()
    conn.close()
    return user

def get_user_by_id(id):
    conn = sqlite3.connect('db/users.db')
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()

    sql = 'SELECT * FROM users WHERE id = ?'
    cursor.execute(sql, (id,))
    user = cursor.fetchone()

    cursor.close()
    conn.close()
    return user

def add_user(user):
    conn = sqlite3.connect('db/users.db')
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()

    success = False
    sql = "INSERT INTO users(nome, cognome, email, password) VALUES(?, ?, ?, ?)"
    try:
        cursor.execute(sql, (user['name'], user['surname'], user['email'], user['password']))
        conn.commit()
        success = True
    except Exception as e:
        print('ERROR', str(e))
        conn.rollback()

    cursor.close()
    conn.close()
    return success

→ models.py [per la classe User, che definisco con default di UserMixin]

from flask_login import UserMixin

class User(UserMixin):
    def __init__(self, id, name, surname, email, password):
        self.id = id
        self.name = name
        self.surname = surname
        self.email = email
        self.password = password

```

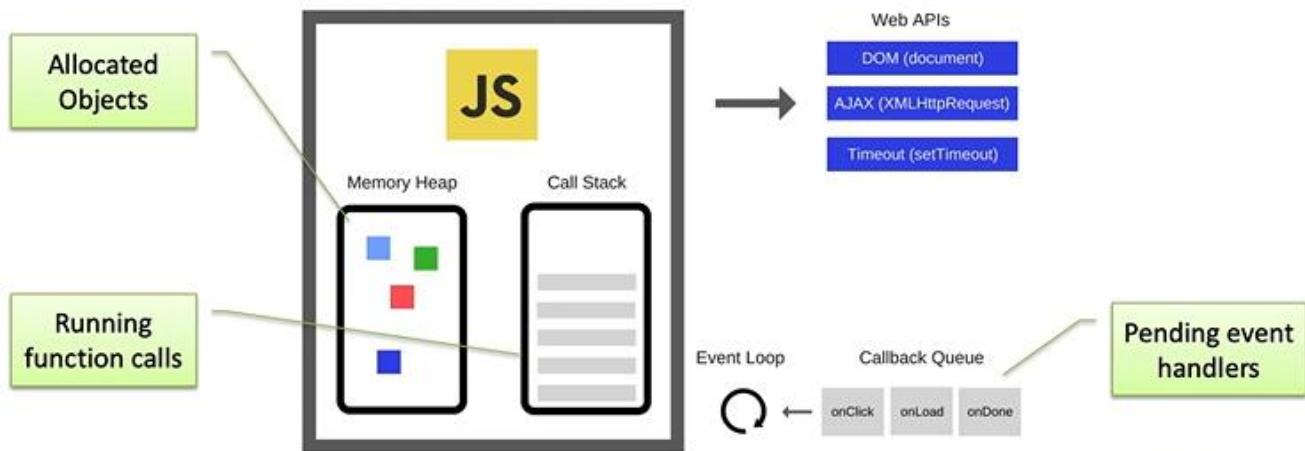
8) JAVASCRIPT

JavaScript è l'unico linguaggio di programmazione nativamente supportato dai browser (può essere anche eseguito su nostro computer mediante NodeJS, ma in questo corso lo useremo solo nei browser); non ha niente a che vedere con Java. Scrivendo in JavaScript bisogna capire se il metodo che stiamo utilizzando è supportato dai principali browser; inoltre, possiede completa retrocompatibilità con le versioni precedenti (infatti se inseriamo la stringa “`use strict`”; all'inizio del nostro programma JavaScript, possiamo disabilitare alcuni metodi delle versioni di JavaScript precedenti [in questo modo si tolgonono delle porcherie]).

⚠ Nella modalità ispezione posso scrivere codice JavaScript anche nella console del browser! È scritto in UNICODE (supporta anche le emoji), i “;” non sono obbligatori ma noi li useremo sempre e la sintassi è C-like (anche i COMMENTI si scrivono con `/*commento*/` [multiriga] oppure `//commento` [riga singola]).

Il codice JavaScript si può importare nel documento HTML con una scrittura diretta nell'HTML (`<script> Codice JavaScript </script>`) oppure da un file.js (`<script src="file.js"></script>`). Noi useremo la soluzione dal file esterno e scriveremo `<script src="file.js"></script>` al fondo del body (ovvero prima di chiudere il tag `<body>`); oppure possiamo comunque scrivere l'importazione nel `<head>` attraverso la stringa `<script defer src="file.js"></script>`.

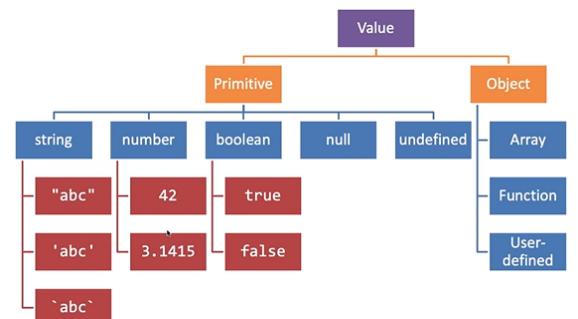
JavaScript si basa su eventi asincroni, ma dato che è single-threaded (un solo processo alla volta), funziona con una coda di eventi (EVENT LOOP):

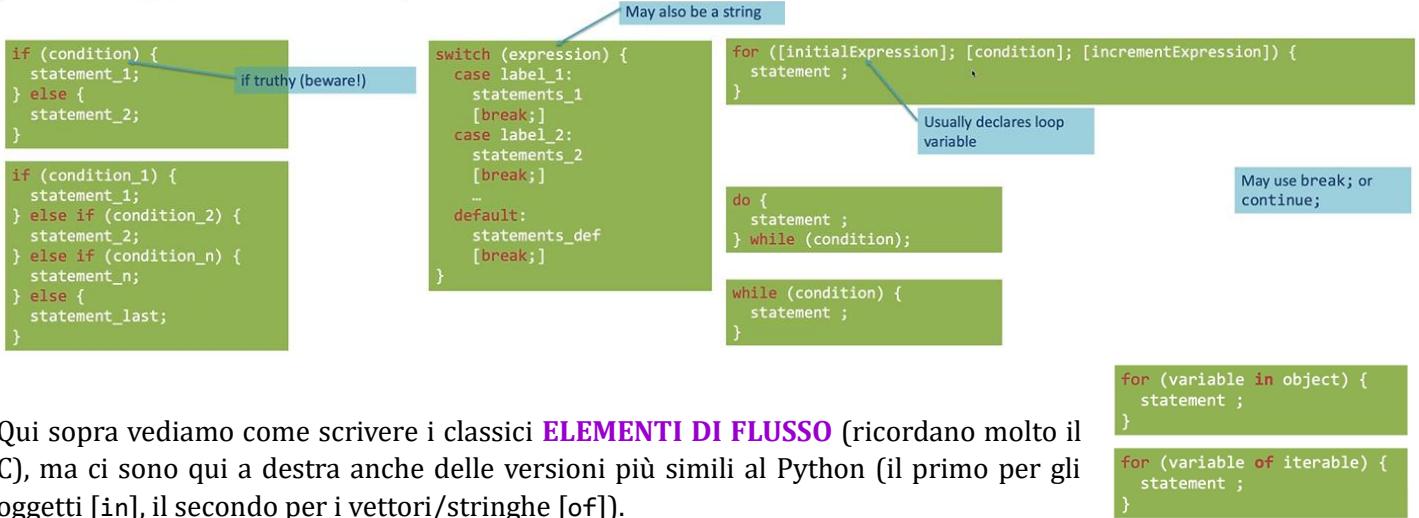


A livello di variabili e tipi è molto simile al python più che al C, ovvero le **VARIABILI** non hanno un tipo con cui vengono dichiarate ma **possono contenere ogni tipo** (il tipo è dato dal valore che troviamo al loro interno). Non c'è distinzione tra `int` e `float` (c'è conversione automatica a seconda dell'operazione); tutte le variabili dichiarate e non inizializzate (non assegno valore) contengono il valore **`undefined`**. Altro valore importante è il **`null`** (valore vuoto) e il **`NaN`** (Not A Number, ovvero un numero che rappresenta un errore aritmetico). Per inizializzare una variabile posso fare **variabile=valore**, ma sconsigliato (vietato in strict mode); è meglio usare:

- **let** variabile = valore → posso riassegnare il valore, ma non ridichiarare la variabile (anche se posso usare le graffe `{...}` per creare un nuovo scope e quindi ridichiare la variabile);
- **const** variabile = valore → non posso riassegnare né ridichiare la variabile;
- **var** variabile = valore → posso sia riassegnare che ridichiare la variabile.

⚠ Valgono le solite regole di assegnazione strutturata con altre operazioni (ovvero `+=` etc...), i confronti e le operazioni; per confrontare gli oggetti non useremo `==` oppure `==`, ma useremo altri comparatori.





Qui sopra vediamo come scrivere i classici **ELEMENTI DI FLUSSO** (ricordano molto il C), ma ci sono qui a destra anche delle versioni più simili al Python (il primo per gli oggetti [in], il secondo per i vettori/stringhe [of]).

Per quanto riguarda i **VETTORI** (arrays) in JavaScript troviamo un sacco di proprietà legate ad essi; **si creano** come in python con le parentesi quadre [] (oppure uso **vettore = Array.of(1,2,3)**) e troviamo il metodo **.length** (che ci fornisce la sua lunghezza). Posso **riempire** gli array con **.push()**, **.pop()**, **.unshift()** e **.shift()** oppure usando direttamente inserendo elementi con **vettore[indice] = valore**, selezionando l'indice dove non abbiamo ancora elementi (si allungherà da solo il vettore). Posso **copiare** tra loro vettori con **vettore2 = Array.from(vettore1)** [e non con **vettore2 = vettore1**, dove viene solo copiato il puntatore]. E molti altri metodi consultabili. Anche le **STRINGHE** hanno i soliti metodi (es. **toLowerCase()**), ma troviamo anche la possibilità di inserire una stringa in un'altra (es. 1^ riga → **let name = "Bill";** 2^ riga → **let hello = `Hello \${ name }.`;**).

Abbiamo poi gli **OGGETTI**, che possiamo creare anche senza definire una classe (quindi diverso da Java); infatti ogni metodo/funzione dell'oggetto è “public”. Gli oggetti in JavaScript sono **come i dizionari in Python** (es. infatti anche qui si accede con **book[“author”]** oppure con **book.author**). Se vogliamo definire una proprietà dell'oggetto con un nome con uno spazio, dobbiamo usare le virgolette; così come accedo ad una proprietà dell'oggetto, allo stesso modo la **aggiungo** (es. **book[“chapters”] = 10** aggiunge il campo chapters all'oggetto). **Copiare** oggetti si fa con **let oggetto2 = Object.assign({}, oggetto1);** In JavaScript come in python troviamo l'operatore **in**.

Abbiamo poi le **FUNZIONI**, che come in Python sono anch'esse degli oggetti; qui le funzioni possono anche avere un numero indefinito di parametri se inserisco ...arr come parametro al fondo (es. **function sum (par1, par2, ...arr)**). Il **metodo preferibile** per dichiarare le funzioni in JS è quello qui a destra sotto (con la **costante**).

```

let point = { x: 2, y: 5 };

let book = {
    author : "Enrico",
    title : "Learning JS",
    for: "students",
    pages: 520,
};

```

```

function square(x) {
    let y = x * x ;
    return y ;
}

const fn = function(params) {
    /* do something */
}

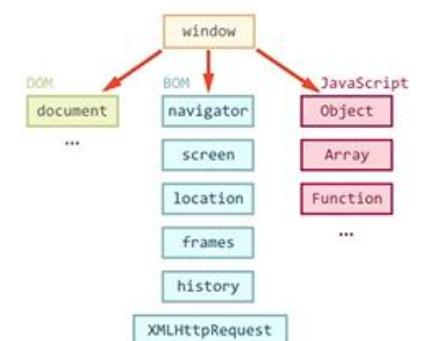
```

Per quanto riguarda la gestione delle **DATE**, JavaScript ci fornisce l'**oggetto Date** (basato sui millisecondi passati dal 01/01/1970 [Unix Epoch]), che però dovremmo **gestire a livello di timezone** (fusi orari); per questo si preferisce **usare delle librerie aggiuntive** (noi useremo Day.js) che ci facilitano anche la manipolazione e il confronto delle date [per importare una libreria in JavaScript dobbiamo importare il file **Day.js** in un tag **<script>** del nostro documento HTML (quindi o scarichiamo il file Day.js e lo mettiamo nel nostro progetto ogni volta oppure [preferibile] **<script src="https://cdn.jsdelivr.net/npm/dayjs@1/dayjs.min.js"></script>**).

⚠ Ora vediamo la parte che ci interessa di JavaScript, ovvero **JAVASCRIPT NEL BROWSER** (il codice JS viene eseguito nel **sandbox del browser**, ovvero

un ambiente protetto dove possiamo accedere alla libreria standard JS, al BOM [Browser objects] e al DOM [Document objects], tutti contenuti nell'**oggetto window** [con tutte le su proprietà]). Oltre ai cookies, sessioni e server, i dati possono anche essere memorizzati nel **window.localStorage** (più sicuro dei cookies e più capiente).

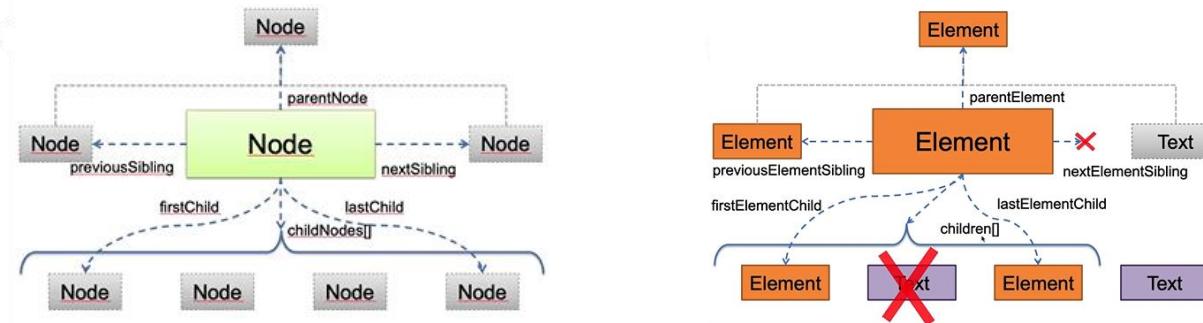
Dato che gli elementi del DOM vengono manipolati come liste di nodi, esiste un tipo chiamato **NodeList** (un vettore di nodi), gestibile come i classici vettori.



Per trovare/richiamare i vari elementi del DOM ci sono dei metodi appositi:

- `document.getElementById(value)` → ritorna il nodo con id=value;
- `document.getElementsByTagName(value)` → ritorna la nodelist degli elementi con un certo <tag>;
- `document.getElementsByClassName(value)` → ritorna la nodelist degli elementi con class=value;
- `document.querySelector(css)` → ritorna il 1° nodo che trovo che ha un match con il selettore css;
- `document.querySelectorAll(css)` → ritorna la nodelist degli elementi con un match con il selettore css.

Possiamo anche navigare tra i nodi muovendoci tra nodi padre e figlio (**parent** e **child**), così come navigare nell'albero degli elementi HTML:



Manipolando il DOM, abbiamo che i <tag> del documento HTML diventano le proprietà dell'oggetto del DOM (es. `<body id="page">` nell'HTML diventa `document.body.id="page"` nel JavaScript); dunque poi possiamo lavorare sugli attributi dell'oggetto del DOM con `element.hasAttribute(name)`, `element.getAttribute(name)`, `element.setAttribute(name, value)`, `element.removeAttribute(name)`, `element.attributes`, `element.matches(css)`. Inoltre possiamo anche creare elementi con `document.createElement(tag)` oppure creare un text node con `document.createTextNode(text)` [ricordarsi però `document.body.appendChild(div)` perché va aggiunto da qualche parte nell'albero HTML oltre ad essere creato].

Allo stesso modo posso operare sulle classi degli elementi con `element.classList.add(name)`, `element.classList.remove(name)`, `element.classList.toggle(name)`, `element.classList.contains(name)`; esiste anche `element.style` ci permette di visualizzare lo stile css di un elemento (e tanti altri metodi analoghi).

Molto importante in JavaScript è la **GESTIONE DEGLI EVENTI** che possono avvenire su un elemento (es. ci passo sopra, lo clicco, scorro la pagina etc...) o sull'intera pagina; l'evento è caratterizzato da il **TIPO** di evento generato (click, scroll ...) e dall'elemento che lo scatena (il **TARGET**): noi dovremo definire delle funzioni "event-listeners", ovvero che captano gli eventi sui vari elementi e li gestiscono (`target.addEventListener('tipo', event =>{})`).

```
const link = document.getElementById('my-link')
link.addEventListener('mousedown', event => {
  // mouse button pressed
  console.log(event.button) // 0=left, 2=right
})
```

⚠ Posso anche prevenire il comportamento di default di alcuni eventi (es. clicco sul link e vengo reindirizzato) con `event.preventDefault()`!

Alcuni eventi si verificano continuamente (es. il movimento del mouse o lo scroll) [eventi continui], quindi dobbiamo usare dei "timeout" per usarli. Ci sono dei metodi per lavorare direttamente sull'HTML con JavaScript (vedendo il codice HTML come stringa), come `element.innerHTML`.

In JavaScript, grazie alla proprietà **dataset**, posso inserire negli elementi HTML un attributo del tipo **data-qualcosa="valore"** per creare degli attributi di tag personalizzati (guardare su YT per capire bene).

⚠ Come il CSS, anche i file JavaScript vanno messi nella cartella **static**!

ESERCIZIO – JavaScript Base (esempio di file JS)

```
'use strict';

/* il COSTRUTTORE (come in Java) */
function Exam(code, name, credits, date, score) {
  this.code = code;
  this.name = name;
  this.credits = credits;
  this.date = dayjs(date);
```

```

        this.score = score;
    }

/* la lista degli Exam (costruttore della list) */
function ExamList() {
    this.list = [];

    this.init = () => {
        this.list.push(
            new Exam('16ACF', 'Analisi I', 10, '2021-02-01', 28),
            new Exam('15AHM', 'Chimica', 8, '2021-02-15', 21),
            new Exam('14BHD', 'Informatica', 8, '2021-02-06', 30),
        );
    };

    this.getAll = () => {
        return this.list;
    }
}

// metodo più rapido
function createTableRow2(exam) {
    return `<tr>
        <td>${exam.date.format('YYYY-MM-DD')}</td>
        <td>${exam.name}</td>
        <td>${exam.credits}</td>
        <td>${exam.score}</td>
        <td><button id="${exam.code}" class="btn btn-success">+</button></td>
    </tr>`;
}

function createTableRow(exam){
    const tr = document.createElement('tr'); // creo riga della tabella

    const tdDate = document.createElement('td'); // creo singoli elementi della riga
    tdDate.innerText = exam.date.format('YYYY-MM-DD'); // formatta la stringa "data"
    tr.appendChild(tdDate);

    const tdName = document.createElement('td');
    tdName.innerText = exam.name;
    tr.appendChild(tdName);

    const tdCredits = document.createElement('td');
    tdCredits.innerText = exam.credits;
    tr.appendChild(tdCredits);

    const tdScore = document.createElement('td');
    tdScore.innerText = exam.score;
    tr.appendChild(tdScore);

    // creo il bottone per togliere esami
    const tdAction = document.createElement('td');
    const button = document.createElement('button');
    button.id = exam.code;
    button.className = 'btn btn-danger';
    button.innerText = 'X';
    tdAction.appendChild(button);
    tr.appendChild(tdAction);

    tdAction.addEventListener('click', e =>{
        tr.remove();
        console.log(e.target.id);
    })
}

return tr;
}

function fillExamTable(exams) {
    const examTable = document.querySelector('#exam-table');

```

```
// equivalente a -> const examTable = document.getElementById('exam-table');
for(let exam of exams){
    const examEl = createTableRow2(exam);
    // metodo classico (dove aggiungo i nodi):
    //examTable.prepend(examEl);
    // metodo 2 (passo il codice HTML con le stringhe literal `${exam.name}`):
    examTable.insertAdjacentHTML('afterbegin', examEl);
}
}

/* MAIN */
const examList = new ExamList();
examList.init();
const exams = examList.getAll();
fillExamTable(exams);
```

⚠ Nel progetto d'esame dovremo unire anche JS a tutto quello visto finora (con Flask, Bootstrap, HTML)!

⚠ Guarda video su domande d'esame in Guarda più tardi su YT!