



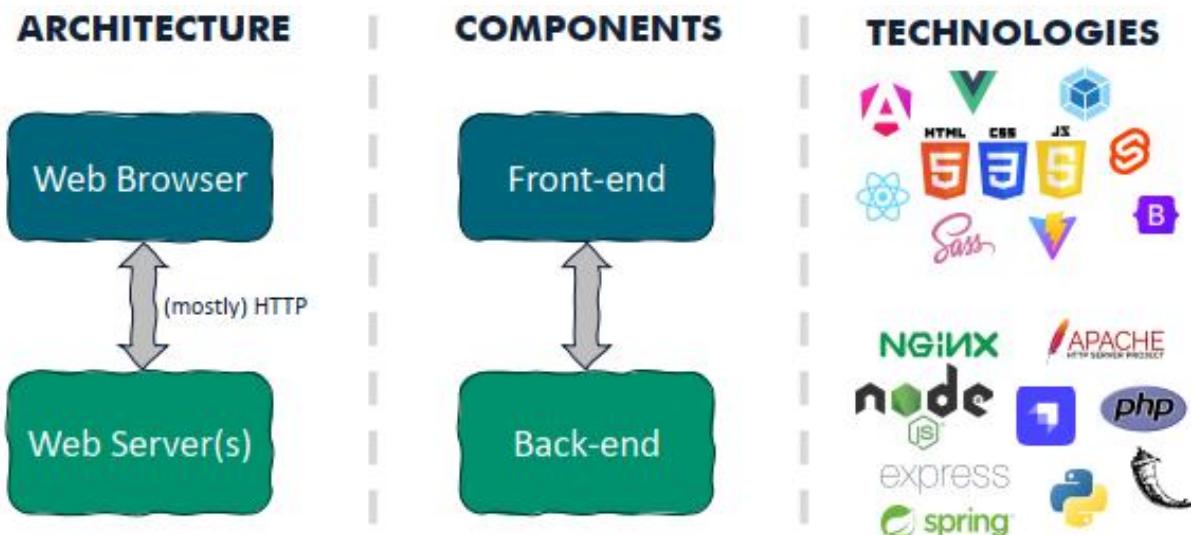
Riassunti del corso di
TECNOLOGIE WEB

A.A. 2023/2024 - Docente: L.L.L. Starace

a cura di F. Formicola

Si consiglia vivamente di fare uso di questi appunti solo dopo aver seguito il corso

Applicazioni Web: pila completa



World Wide Web

Sistema di **documenti ipertestuali** interconnessi tra loro tramite **hyperlinks**.

→ Documenti contenenti caratteri e links che indirizzano ad altri documenti.

Le componenti principali del WWW sono **HTTP** ed **HTML**:

HTTP (Hypertext Transfer Protocol): Protocollo al livello di applicazione costruito sulla base del **TCP/IP**.

I Client effettuano domande ai **Server**, i quali rispondono. Le Risorse sono identificate tramite degli **URLs**

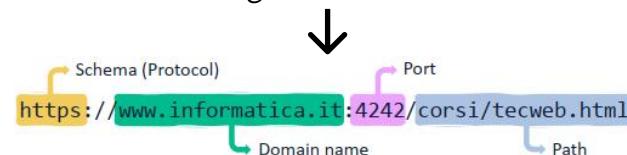
Tipologia di richieste HTTP:

GET: Recupera (la rappresentazione di) una risorsa.

POST: Carica nuovi dati alla risorsa specificata.

PUT: Sostituisce la risorsa corrente col carico inviato.

DELETE: Elimina la risorsa specificata.



Uno dei modi per passare informazioni aggiuntive attraverso richieste e risposte HTTP è quello degli **Headers**: un insieme di coppie chiave-valore nella forma HEADER_NAME: VALUE.

Per specificare se una richiesta è stata (correttamente) esaudita o meno, le risposte sono correlate da uno **Status Code**.

Informational (100-199)	Success (200-299)	Redirection (300-399)	Client Error (400-499)	Server Error (500-599)
100 Continue	200 OK	301 Moved	400 Bad Req. 403 Forbidden 404 Not Found	500 App. Error 503 Unavail.

HTTP è un protocollo **stateless**: non tiene traccia delle richieste fatte in passato dai clients. Per rimediare, si possono usare tecnologie apposite come i **Cookies**: delle stringhe che, inserite negli headers di una richiesta/risposta, forniscono una sorta di autenticazione del client.

HTML (Hypertext Markup Language): Linguaggio di **annotazione** per ipertesti che consente di rappresentarli nel Web.

I documenti definiti usando HTML possono essere correlati di informazioni aggiuntive che ne permettono la modifica e la personalizzazione.

HTML: Hypertext Markup Language

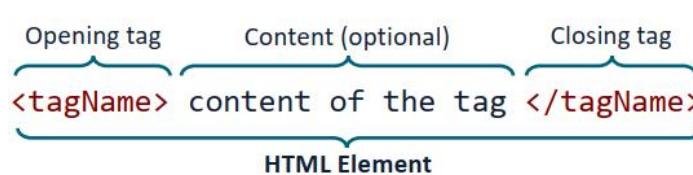
Lezione 2

Un **Browser Web** mostra **Documenti** descritti tramite HTML; un insieme di documenti interconnessi si dice **Web App**.

I documenti HTML sono arricchiti da una serie di annotazioni (**tags**) che consentono di specificarne la struttura, la formattazione, e le relazioni tra le parti.

I tag possono (eventualmente) anche contenere **attributi**, scritti sotto forma di coppie chiave-valore.

Ogni documento HTML inizia con la dichiarazione **<!DOCTYPE HTML>**, che serve a comunicare al client il tipo di documento da aspettarsi.



<tagName attribute1="value" attribute2>

Tipologia di Tags HTML:

<html>: rappresenta l'intero documento, e contiene sempre una **<head>** ed un **<body>**.
<head>: contiene i **metadati** del documento, ossia informazioni sul documento corrente che sono utili (oltre che all'utente) soprattutto ai Browser per operazioni di ricerca; deve necessariamente contenere un **<title>**.
<body>: rappresenta il contenuto effettivo della pagina.
<title>: nome della pagina web.
<h1> to **<h6>**: sono **headings**, atti a rappresentare titoli e sottotitoli.
<p>: rappresenta un paragrafo, che di default inizia su una nuova riga.
<!-- content -->: **commenti**, il cui contenuto è ignorato dal Browser.

Tags per la semantica del testo:

****: corsivo.
****: grassetto.
</br>: line break: va a capo.
<abbr title="description">: definisce acronimi e abbreviazioni.
****: elemento che è stato cancellato dal documento.
<ins>: elemento che è stato inserito nel documento.

Anchors (per la definizione di hyperlink):

<a>: definisce un hyperlink, e l'attributo **href="path"** definisce l'URL target. L'**URL** può essere **assoluto** (include schema ed hostname, e contiene tutte le informazioni necessarie per raggiungere la risorsa) o **relativo** (schema ed hostname vengono omessi in quanto vengono recuperati dal contesto corrente, e viene specificato solo il path). I path relativi possono anche contenere dei segmenti **dots**: **'./'** rappresenta la directory corrente, e **'../'** rappresenta la directory padre.
Conviene utilizzare **URLs relativi** per collegare risorse della stessa applicazione Web, mentre si è tenuti ad usare **URLs assoluti** per collegare risorse esterne.
L'attributo **target** specifica dove aprire la risorsa collegata: il valore può essere **"self"** per aprirla nella pagina corrente oppure **"_blank"** per aprirla in una nuova finestra.

Altri tags:

<table>: tabella contenente un set di righe dette **<tr>** ed eventualmente una **<caption>** che la descriva.
<tr>: può contenere una o più **<td>** (table data cell) e **<th>** (table headers).
<td> e **<th>**: sono il contenuto da mostrare nelle rispettive celle della tabella.
****, ****, **<dl>**: rispettivamente per liste ordinate, liste puntate e liste descrittive (**<dt>** per i termini e **<dd>** per le descrizioni).
****: list item, ossia un oggetto della lista.
****: per inserire un'immagine nel documento HTML. Correlato dagli attributi **src** per specificare l'URL dell'immagine, **alt** per inserire una descrizione alternativa dell'immagine, e **width** e **height** per specificarne le dimensioni in pixel. Se l'immagine proviene dall'esterno della pagina, il Browser effettua una richiesta HTTP GET per il retrieve della risorsa.

Entities

Necessarie per scrivere all'interno di un documento HTML quei caratteri che sono parte della sintassi del linguaggio.

Hanno la forma di **&entity_name** oppure **&#entity_number**

Result	Description	Entity Name
Non-breaking space		
<	Less than	<
>	Greater than	>
&	Ampersand	&
"	Double quote	"
'	Single quote (apostrophe)	'
©	Copyright	©

Attributi Globali

Attributi validi per tutti gli elementi del documento e non solo per quelli rinchiusi nello specifico tag.

id: specifica un identificativo **univoco** (in quel documento) per un elemento.

lang: specifica la lingua del contenuto del documento.

style, class: per modificare per lo stile del documento.

Forms

<form>: serve per raccogliere gli input dell'utente, i quali vengono tipicamente spediti ad un server che li processa.

Può contenere del testo e diverse tipologie di **<input>**.

<label>: serve per attribuire un nome ad uno specifico elemento del form, per usabilità. L'attributo **for** delle label deve corrispondere all'id dell'input corrispondente.

I form possono essere inviati (**submitted**) ad un'entità che saprà come gestirli (**form-handler**) tramite una richiesta HTTP. Per specificare l'URL del form handler si utilizza l'attributo **action**, mentre per specificare il metodo della richiesta si utilizza l'attributo **method** (che è GET di default).

Type	Description
<input type="text">	Displays a single-line text input field
<input type="password">	Displays an input field for passwords (input is hidden with *****)
<input type="number">	Displays an input for numbers
<input type="radio">	Displays a radio button (for selecting one of many choices)
<input type="checkbox">	Displays a checkbox (for selecting zero or more of many choices)
<input type="button">	Displays a clickable button

A seguito della submission, gli input sono rappresentati come una serie di coppie chiave-valore della forma **name1=value1&...&nameN=valueN**, in cui ogni nome corrisponde ad un elemento del form, ed il valore corrisponde a qual era il valore inserito dall'utente al momento dell'invio.

Utilizzando il metodo **GET**, gli input vengono concatenati all'URL dell'handler nella richiesta HTTP come **Query Parameters**, separati gli uni dagli altri da '&'.

Utilizzando il metodo **POST**, gli input vengono inviati all'handler all'interno del **Body** della richiesta HTTP.

Se l'utente inserisce dei caratteri speciali all'interno dei valori degli input, questi vengono codificati nell'URL con la forma '%XX', in cui le X corrispondono a dei numeri esadecimali.

```
GET /handler.html?msg=Hello!&num=42 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: text/plain
Accept-Language: en-us
Connection: keep-alive
```

```
GET /handler.html HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: text/plain
Accept-Language: en-us
Connection: keep-alive
msg=Hello!&num=42
```

Lavorare sugli input

Ci sono diverse tipologie di input, ad esempio: **checkbox**, **radio** (per radio buttons), **date**, **time**, **select** (per liste dropdown), **color**. È inoltre possibile raggruppare diversi input all'interno di uno o più tag **<fieldset>**, ai quali si può anche attribuire un titolo (attraverso il tag **<legend>**) per una maggiore comprensibilità del codice e del form in sé.

Organizzare il contenuto del documento

Il contenuto di una pagina web può essere raggruppato in parti attraverso delle divisions o dei tag semantici.

<div>: serve ad effettuare un semplice raggruppamento di elementi che hanno una qualsiasi relazione tra loro.

I **tag semantici**, invece, descrivono il significato del contenuto ai Browser, ai developer ed eventualmente ad altri software.

Alcuni di questi sono:

<nav>: contiene hyperlink per la navigazione.

<main>: indica il contenuto principale

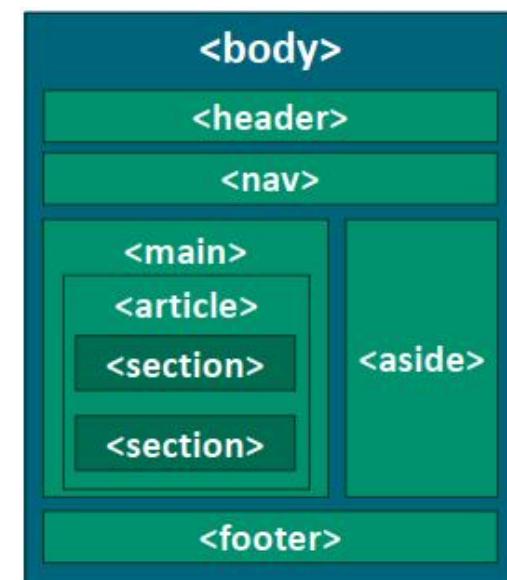
<article>: utilizzato per contenuti autonomi ed indipendenti.

<aside>: per elementi legati in maniera tangenziale.

<header>: indica l'intestazione.

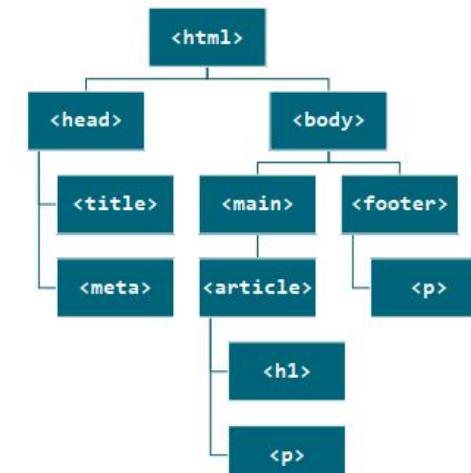
<footer>: indica il piè di pagina.

<section>: indica sessioni a sé stanti all'interno di altri containers.



Tramite il raggruppamento è possibile quindi vedere il contenuto di un documento HTML sotto forma di un **ALBERO**:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>The Book of Programming</title>
  <meta charset="UTF-8">
</head>
<body>
  <main>
    <article>
      <h1>Title</h1>
      <p>Body</p>
    </article>
  </main>
  <footer>
    <p>&copy; Web Technologies 2024</p>
  </footer>
</body>
</html>
```



Browser Dev Tools

A supporto dei web developer ci sono molte feature all'interno dei Browser, accessibili premendo il tasto **F12**.

Tra le più importanti: la possibilità di ispezionare il documento HTML, l'analisi delle richieste/risposte HTTP coinvolte, la misurazione delle performance ed il debugging.

CSS: Cascading Style Sheets

Lezione 3

Un **linguaggio dichiarativo** basato su regole che specifica il modo in cui un documento deve essere presentato agli utenti

Uno **stylesheet** (foglio di stile) è un insieme di **regole**, definite nel seguente modo:

- un **selettor** che specifica su quali elementi HTML le regole saranno applicate;
- un insieme di **dichiarazioni** scritte nella forma **proprietà: valore** che specifica lo stile da applicare agli elementi.

```
selector {
  property: value;
  property: value;
}
```

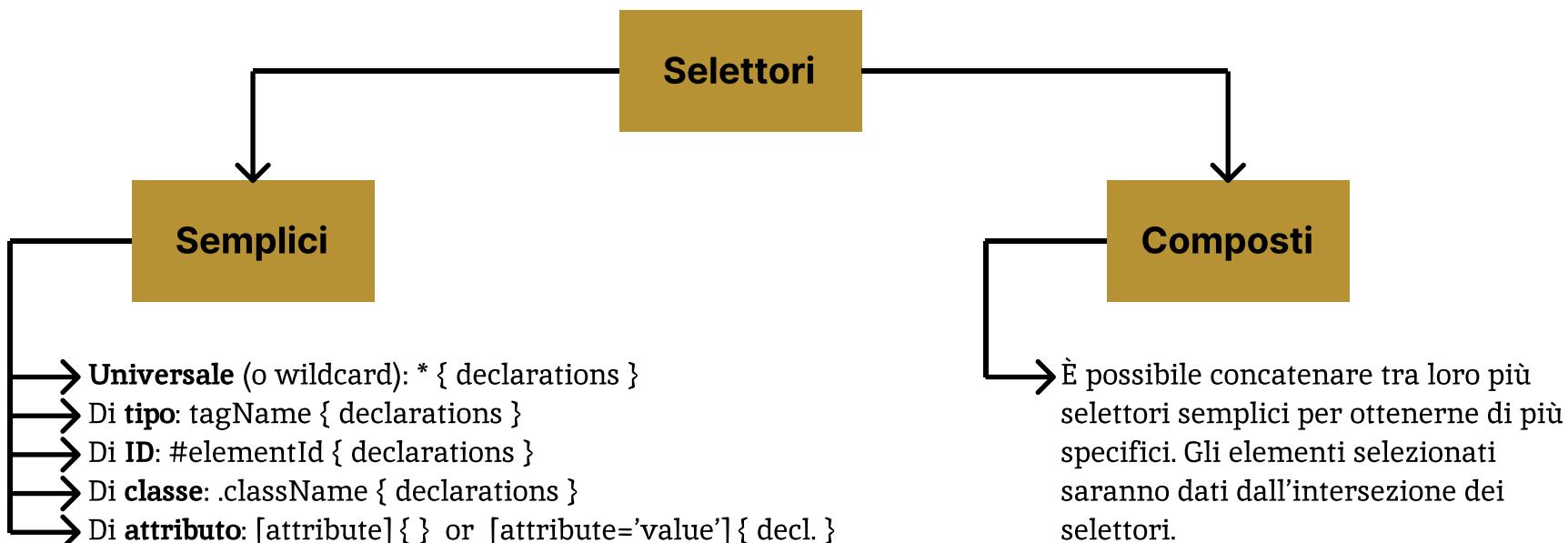
Uno stile può essere inserito all'interno di un documento in diversi modi:

- usando `<link>` all'interno del tag `<head>`, con attributi `rel="stylesheet"` e `href="style.css"`;
- scrivendo direttamente regole CSS nel tag `<style>` all'interno del tag `<head>`;

è inoltre possibile modificare **inline** lo stile di singoli elementi del documento HTML utilizzando l'attributo `style` con, come valore, una sequenza di dichiarazioni separate da `'.'`.

Selettori

Specificano a quale elemento si applica una regola CSS.



Operatori aggiuntivi che consentono il matching parziale
dei valori degli attributi:

- `*='value'`: contiene 'value';
- `^='value'`: inizia con 'value';
- `$='value'`: termina con 'value';

Ci sono **combinatori** che consentono di selezionare elementi in base alla loro posizione nel documento HTML, che ricordiamo, può essere visto come un albero:

Selettore di discendenti (spazio): selectorA selectorB

- Seleziono tutti gli elementi che matchano il selettore B, e, contemporaneamente, sono **discendenti** di un elemento che matcha il selettore A.

Selettore di figli (>): selectorA > selectorB

- Seleziono tutti gli elementi che matchano il selettore B, e, contemporaneamente, sono **figli diretti** di un elemento che matcha il selettore A.

Selettore di fratelli adiacenti (+): selectorA + selectorB

- Seleziono l'elemento che matcha il selettore B, e, contemporaneamente, è **fratello destro adiacente** di un elemento che matcha il selettore A. Nota bene: in CSS ed HTML non si può navigare all'indietro nella pagina.

Selettore generale di fratelli (~): selectorA ~ selectorB

- Seleziono tutti gli elementi che matchano il selettore B, e, contemporaneamente, vengono dopo un elemento che matcha il selettore A.

Pseudo-Classi

Iniziano per `:` e sono dei selettori che consentono di dare uno stile ad elementi in base allo **stato** di questi in un determinato momento, dovuto all'interazione con l'utente o con altre componenti del documento:

Stati interattivi: dovuti all'interazione con l'utente.

- `:hover` : seleziona l'elemento su cui il puntatore è posizionato sopra.
- `:active` : imposta la presentazione di un elemento mentre questo ha un'interazione attiva con l'utente.
- `:focus` : imposta la presentazione di un elemento mentre questo è stato selezionato dall'utente.

Stati storici: dovuti ad eventi passati.

- `:link` : seleziona link che non sono stati ancora visitati.
- `:visited` : seleziona link che sono già stati visitati.

Stati di Form: dovuti allo stato degli elementi all'interno di un Form.

- `:disabled` : seleziona gli elementi disabilitati.
- `:invalid` : seleziona gli elementi invalidi.
- `:checked` : seleziona elementi di una checkbox o di un radio button che sono stati selezionati dall'utente.

Relazioni di posizione: dovuti alla posizione degli elementi all'interno del documento HTML.

- `:first-child` e `:last-child` : selezionano, rispettivamente, il primo e l'ultimo figlio tra una lista di fratelli.
- `:only-child` : seleziona elementi che non hanno fratelli.
- `:first-of-type` e `:last-of-type` : selezionano, rispettivamente, il primo e l'ultimo figlio tra una lista di fratelli ma considerando solo elementi dello stesso tipo.
- `:nth-child(n)` e `:nth-of-type(n)` : selezionano l'n-esimo elemento da una lista di fratelli. L'indexing in CSS parte da 1.

Pseudo-Elementi

Servono a selezionare delle parti interne ad alcuni elementi, consentendo di modificarle, senza aggiungere ulteriore markup all'interno del documento HTML. La sintassi è **selector::pseudo-element**. Gli pseudo-elementi sono i seguenti:

- **::first-letter** : seleziona la prima lettera del contenuto di un elemento a livello di blocco.
- **::first-line** : seleziona la prima riga del contenuto di un elemento a livello di blocco.
- **::selection** : seleziona il contenuto dell'elemento attualmente selezionato dall'utente.
- **::before** : crea un elemento che sarà il primo figlio dell'elemento selezionato.
- **::after** : crea un elemento che sarà l'ultimo figlio dell'elemento selezionato.

L'Algoritmo CASCADE

Algoritmo che produce in output la risoluzione (ossia, l'ordine di esecuzione) dei **conflitti** dati in input (dovuti a più regole che potrebbero essere applicate agli stessi elementi), e lo fa in base a quattro aspetti chiave, in ordine di importanza:

1. Origine ed Importanza:

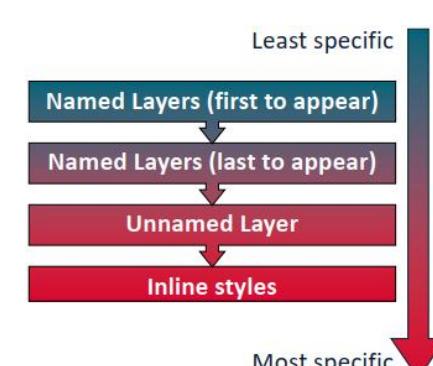
Si occupa di verificare la provenienza della regola, ossia se questa deriva da uno stile dello **User Agent**, dell'**Utente**, o se è **Autoritario** (scritto manualmente dal developer).

Inoltre, una regola che può essere aggiunta per dare ulteriore importanza ad una proprietà CSS è **!important**: questa viene aggiunta alla fine della dichiarazione di una proprietà, e la rende più specifica rispetto a quelle non esplicitamente importanti. L'obiettivo è quello di andare incontro all'accessibilità.



2. Livelli:

Le regole dei fogli di stile autoritari possono essere raggruppate in **livelli** a cui è possibile dare un nome; inoltre, regole dettate nel tag `<style>` o importate da fogli di stile esterni, vengono dette "senza nome". Ancora, gli stili descritti **inline** appartengono ad un ulteriore livello e sono quelli che hanno una priorità maggiore.



3. Specificità:

Se due regole sono in conflitto, appartengono allo stesso **bucket** di origine-importanza, ed appartengono allo stesso livello, allora si passa al concetto di specificità. L'idea è che debba vincere il selettori più specifico, e per calcolarlo si utilizza una tripla (**A, B, C**) in cui:



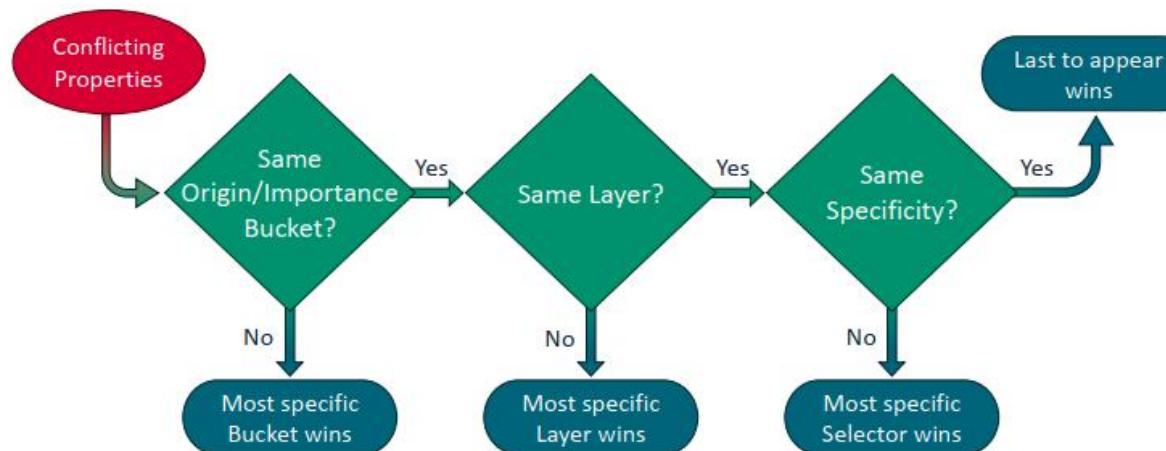
- I selettori universali vengono ignorati;
- Si conta il numero di **selettori di ID** (=A);
- Si conta il numero di **selettori di classe, attributo, o pseudo-classe** (=B);
- Si conta il numero di **selettori di tipo o pseudo-elementi** (=C);

Considerando la tripla in ordine, il selettori con il valore più alto (da sinistra verso destra) vince. In caso la tripla sia uguale, i selettori pareggiano.



4. Posizione ed ordine di apparenza della regola:

Quando due o più regole dello stesso foglio di stile, si trovano nello stesso bucket di origine-importanza, sono allo stesso livello ed hanno anche la stessa specificità, si considera semplicemente l'**ordine** con il quale queste appaiono nel foglio di stile: la regola che appare per ultima è la vincitrice.



Il tool **Inspector** dei Browser Web Tools consente di visualizzare le regole dalla più specifica (in alto) alla meno specifica (in basso), mostrando anche (abilitando l'impostazione) gli stili dello User Agent.

Ereditarietà

In CSS, per alcune proprietà come **color**, **font-size**, **font-family**, **font-weight**, **font-style**, esiste il concetto di **ereditarietà**: queste vengono ereditate in base alla discendenza di un elemento.

All'interno dell'algoritmo Cascade, le proprietà ereditate hanno la **specificità più bassa** di tutte.



Lezione 4

CSS: Sizing Units

Si possono utilizzare proprietà CSS per cambiare le dimensioni degli elementi o dei loro contenitori.

Per far ciò, è possibile utilizzare lunghezze **assolute** oppure **relative**.

- **Lunghezze assolute**: definite usando un numero ed un'unità di misura supportata (**px**, **cm**, **in**, **mm**);
- **Lunghezze relative**: definite in relazione alla dimensione dei genitori o dell'intera finestra, come percentage (%), viewport (vw) per la dimensione della finestra del browser, **em** o **rem** per la dimensione rispetto alla grandezza del font;

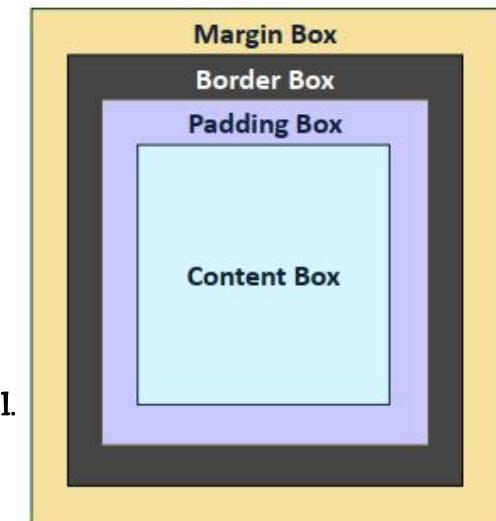
LAYOUTS

In HTML ogni elemento è un **contenitore** (scatola). Il contenuto della scatola è la zona dove vivono i figli dell'elemento.

Il **padding** è lo spazio che separa il contenuto di una scatola dal bordo di questa;

L'elemento **Border** rappresenta i limiti territoriali dell'elemento;

Margin crea dello spazio intorno all'elemento.



La dimensione delle aree può essere definita usando dichiarazioni CSS, ed il loro comportamento è determinato dal loro **Layout**, dal **Contenuto** e dalle **proprietà del Box model**.

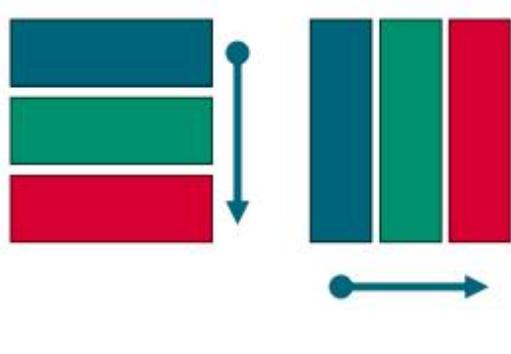
Il layout di default è il **Flow Layout**: gli elementi **Inline** vengono mostrati nella direzione inline (da sinistra verso destra, sulla stessa riga), e gli elementi **Blocks** vengono disposti l'uno sotto l'altro (quindi andando a capo); **inline**, **block** e **none** (quest'ultimo usato per nascondere un elemento dalla visualizzazione) sono valori dell'attributo **display**.

La proprietà **float** può essere usata per far “galleggiare” elementi nella direzione desiderata, facendo sì che i fratelli gli si “avvolgano” attorno (a meno che, su questi, non si applichi la proprietà **clear**).

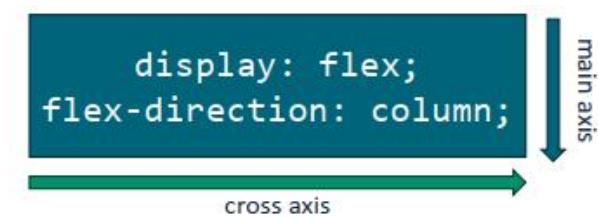
La proprietà **position** cambia il modo in cui un elemento si comporta all'interno del flow del documento. Ci sono diversi valori:

- **static** : di default, l'elemento è esattamente dove dovrebbe essere secondo il normale flow del documento;
- **relative** : l'elemento è in una posizione relativa rispetto alla sua posizione originaria;
- **absolute** : l'elemento è posizionato rispetto al più vicino antenato che ha una position relative;
- **fixed** : l'elemento è posizionato in relazione alla viewport;
- **sticky** : l'elemento è posizionato in base allo scroll dell'utente (fino al raggiungimento eventuale di una certa soglia).

Il **Flex Layout**, designato per layout mono-dimensionali in **orizzontale** o in **verticale**, viene dichiarato con la proprietà **display: flex**.



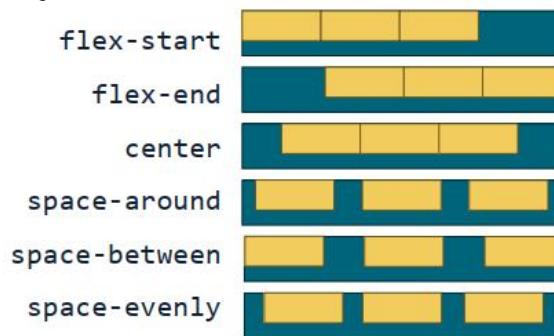
Ci sono degli elementi a livello di blocco, che hanno dei **flex item** come figli. Ogni container flex ha un'asse principale ed un'asse trasversale: la prima è selezionata tramite la proprietà **flex-direction** (che è row di default), la seconda è ortogonale.



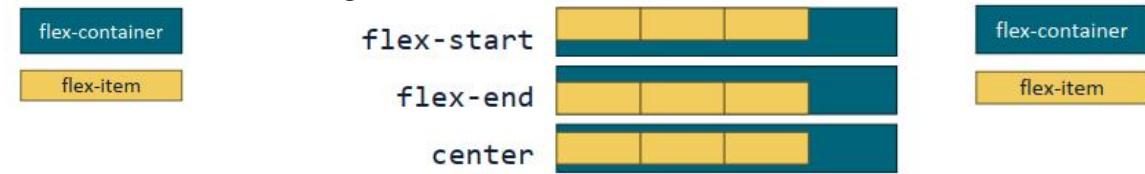
Utili proprietà applicabili sui flex-items sono **flex-grow** (che lo espande fino a raggiungere tutto lo spazio occupabile lungo l'asse principale) e **flex-shrink** (che lo riduce lungo l'asse principale per farlo rientrare nel container) e **flex-wrap** (che controlla eventuali overflow in un flex container).

Ai flex container è possibile applicare proprietà che ne modificano il posizionamento del contenuto, e due tra queste sono **justify-content** (che specifica come lo spazio libero lungo l'asse principale debba essere gestito) e **align-content** (che fa la stessa cosa ma lungo l'asse trasversale).

justify-content:



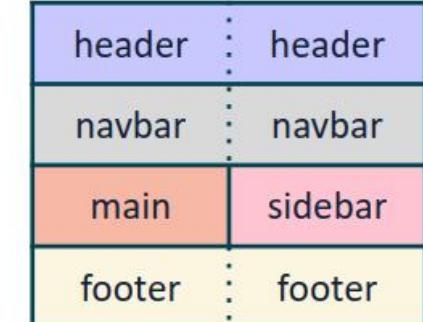
align-content:



Il **Grid Layout**, designato per layout bi-dimensionali con **righe e colonne**, viene dichiarato con la proprietà **display: grid**. I container definiscono **numero e dimensione** delle proprie righe e colonne, ed i loro figli sono detti **grid items**.

Un'unità di misura particolare per i grid layout è la **fr**, che assegna determinate porzioni dello spazio disponibile ai grid items. Si possono assegnare dei **nomi** a delle aree (celle) della tabella grid, ed in queste inserire dei grid items usando la proprietà **grid-area**. In questo modo è possibile creare dei veri e propri **template**.

```
display: grid;
grid-template-columns: 70vw 1fr;
grid-template-rows: auto auto 1fr auto;
grid-template-areas:
  "header header"
  "navbar navbar"
  "main sidebar"
  "footer footer";
height: 100vh;
margin: 0;
```



Media Queries

Le media queries iniziano con la keyword **@media** e consentono di applicare stili CSS agli elementi solo quando il dispositivo che sta visualizzando il contenuto possiede certe caratteristiche (come ad esempio una certa dimensione dello schermo).

Esistono 3 tipi di output per le media queries:

- **print** : specifico per la visualizzazione di pagine in modalità di stampa;
- **screen** : specifico per la visualizzazione di pagine su uno schermo di un dispositivo;
- **all** : si applica a tutti i dispositivi (tipo di default).

Ci si può basare anche su specifiche caratteristiche del dispositivo, dette **Media Features**.

La sintassi completa di una media query è la seguente: **@media media_type and (media_feature: value) and ...**

RESPONSIVE DESIGN

FIXED-WIDTH LAYOUTS: layouts a **dimensione fissa**, in quanto inizialmente molti dispositivi avevano le stesse dimensioni dello schermo.

LIQUID (FLUID) LAYOUTS: layouts a dimensione dinamica ma solo in base alla larghezza delle colonne che era impostata in **percentuale** rispetto allo schermo. Il problema principale è che in casi estremi (schermi troppo schiacciati o troppo allungati) diventavano inutilizzabili.

ADAPTIVE LAYOUTS: layouts un po' più flessibili che utilizzavano **media queries** per scegliere quale **fixed layout** dovesse essere visualizzato a seconda della dimensione dello schermo del device che richiedeva la risorsa.

RESPONSIVE LAYOUTS: sono un mashup di **media queries** e **liquid layouts**. Sono caratterizzati da **containers fluidi**, **media fluidi**, e **media queries**. L'obiettivo è che tutto venga visualizzato in maniera ottimale su ogni dispositivo.

Per ottimizzare il controllo del virtual viewport mechanism si utilizza un **viewport HTML meta tag**:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

che va inserito in ogni pagina affinché questa sia responsive.

Le due regole dicono al browser di far finta che la larghezza della pagina web sia stata scelta appositamente in base a quella del dispositivo, ed inoltre di non effettuare alcuno scaling.

JAVASCRIPT

Linguaggio di programmazione per scripting, debolmente tipato.

Del codice JS può essere inserito all'interno di documenti HTML in maniera **interna** (ossia, contenuto all'interno di un **tag <script>** posto nella head o nel body del documento) oppure **esterna** (tramite un URL che porta ad un file .js esterno, nel seguente modo: `<script src="urlOfScript.js"></script>`). Nel 2009, ECMAScript 5 (ES5) ha apportato importanti modifiche a JS, e per abilitarle nei file moderni si dichiara “**use strict**” all'inizio del documento.

JavaScript supporta i paradigmi **imperativo, funzionale ed orientato agli oggetti**.

Un programma JS è una sequenza di **statements** composto da variabili, operatori, espressioni, parole chiave e commenti.

Variabili

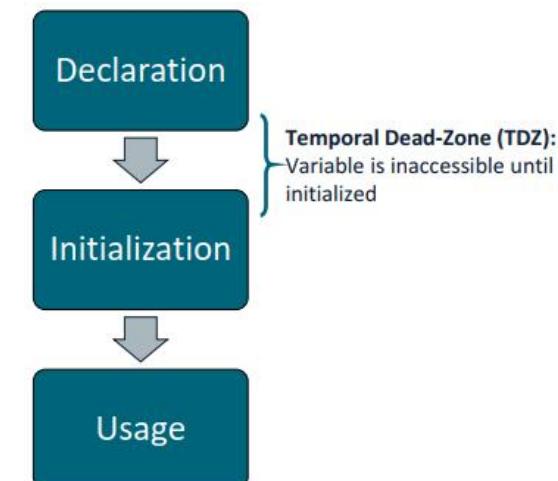
La dichiarazione delle variabili avviene tramite le parole chiave **let** (per variabili standard) e **const** (per variabili che non possono essere modificate). Queste sono inizializzate ad **undefined** di default.

Il **ciclo di vita** di una variabile ha 3 step:

- **dichiarazione** (il nome della variabile è legato allo scope corrente, che può essere globale, di funzione, di modulo, o di blocco);
- **inizializzazione** (viene assegnato un valore alla variabile);
- **utilizzo** (la variabile viene referenziata).

Le variabili sono accessibili soltanto dopo la linea nella quale sono state dichiarate.

Se utilizziamo una variabile che non è presente nello scope del blocco corrente, questa viene ricercata (a catena) nello scope del padre.



In JavaScript esiste il concetto di **Hoisting**: è un comportamento secondo il quale la dichiarazione delle variabili o delle funzioni viene spostata all'inizio del loro scope. Questo non vale però per l'inizializzazione, che avviene alla riga in cui è effettivamente scritta la dichiarazione.

Prima di ECMAScript 6 (2015) le variabili venivano dichiarate con la parola chiave **var**, che effettuava l'hoisting di queste alla funzione più vicina e non a livello di blocco, inizializzandole ad undefined. Oggigiorno non è consigliabile l'utilizzo di var, come anche quello delle **dichiarazioni implicite** (proibite dallo “use strict”: avvengono senza alcuna parola chiave, creando delle variabili globali).

```
// x variable not defined
{
  // Declaration for x is hoisted here
  // TDZ for the x variable
  // TDZ for the x variable
  console.log(x); //raises error
  // TDZ for the x variable
  let x = 1; // TDZ ends, x initialized
  // x variable initialized to 1
  // x variable initialized to 1
}
// x variable not defined
```

! ReferenceError: can't access lexical declaration 'x' before initialization

In JavaScript non esiste il concetto di tipo: ad una stessa variabile può essere assegnato un contenuto diverso ogni volta.

Dati primitivi, operatori di base e di confronto sono molto simili a quelli di Java e C.

Una differenza può essere vista tra gli operatori di uguaglianza debole (`==`), che confronta se due operandi sono uguali di valore, anche se però sono di tipo diverso, e quello di uguaglianza forte (`===`), che confronta anche il tipo degli operandi oltre al valore, fermandosi subito se questo è diverso.

Operator	Description
=	Assignment
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (since ECMAScript 2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

I controlli di flusso (If, if-esle, for, while, do, switch) funzionano esattamente come in Java.

Funzioni

Le funzioni possono essere dichiarate con la sintassi: `function functionName(params) { }`

Se in una chiamata a funzione non viene passato alcun argomento, i **parametri** vengono inizializzati ad `undefined`.

Alternativamente, possono anche essere assegnati ai parametri dei valori di default.

Anche alla dichiarazione di funzioni si applica l'hoisting: si può chiamare una funzione prima della sua effettiva dichiarazione.

```
function greet(name, message="Hello"){ //message has a default value
  console.log(` ${message} ${name}`);
}

greet("Web Technologies");           // Hello Web Technologies
greet("Web Technologies", "Ciao");   // Ciao Web Technologies
greet();                            // Hello undefined
```

Le funzioni, come le variabili, non possono essere referenziate dall'esterno del loro scope; inoltre, è possibile creare funzioni usando espressioni ed **assegnandole ad una variabile**.

Le **funzioni innestate**, ossia quelle create all'interno di un'altra funzione, possono accedere al contesto esterno ad esse, ossia quello della funzione che le ha create.

Tuttavia, anche se la chiamo da un blocco diverso da quello in cui è stata dichiarata, continuerà a vedere le variabili dello scope in cui è presente la sua dichiarazione.

```
// standard function expression
let greet = function(name) {
  console.log(`Hello ${name}!`);
}

greet("Web Technologies"); // output: Hello Web Technologies!
```

```
function getGreeter(message) {
  let sep = ",";
  return function(name) {
    console.log(`${message}${sep} ${name}!`);
  }
}

let helloGreeter = getGreeter("Hello");
helloGreeter("Web"); //Hello, Web!

let howdyGreeter = getGreeter("Howdy");
howdyGreeter("JS"); //Howdy, JS!
```

OGGETTI

Sono dei contenitori di coppie **chiave: valore**.

Si possono usare due sintassi diverse:

- Tramite costruttore: `let a = new Object();`
- Tramite letterale: `let a = {};`

È possibile aggiungere delle **proprietà** ad un oggetto sia alla sua creazione che successivamente; le proprietà hanno una **chiave** (o nome, identificativo) a sinistra, ed a destra, dopo i ':', hanno un **valore**. Sono separate da virgolette.

Si accede (in lettura o scrittura) alle proprietà attraverso la **dot notation** (`objectName.property`), oppure, in caso alcune proprietà dovessero contenere delle espressioni come nome, tramite la **bracket notation** (`objectName["propertyName"]`).

```
let pet = {
  name: "Hannibal",
  age: 7
}

pet.nickname = "the Affable"; // new property
delete pet.age;             // delete property

console.log(pet.name);       // "Hannibal"
console.log(pet.age);        // undefined
console.log(pet.nickname);   // "the Affable"

console.log(pet); // print object in console
```

Quando assegniamo un oggetto ad una variabile, stiamo passando un **riferimento** a questo, e non l'oggetto in sé.

Per effettuare un **clone** di un oggetto, avremmo bisogno di copiare iterativamente ogni proprietà. Si differenzia dunque tra shallow e deep cloning:

- **Shallow Cloning**: se una proprietà di un oggetto è a sua volta un oggetto, durante la clonazione del primo non viene clonato anche il secondo, ma viene copiato solo un riferimento, dunque modifiche al clone modificheranno anche l'oggetto originale;
- **Deep Cloning**: nella copia di un oggetto, si itera su ogni proprietà, e qualora una di queste fosse a sua volta un oggetto, si copiano iterativamente anche le proprietà di quest'ultima.

Metodi

Sono delle **funzioni scritte come proprietà** di un oggetto.

Se un metodo deve modificare altre proprietà dell'oggetto in cui è dichiarato, può farlo attraverso la parola chiave "**this**" (attenzione: non funziona con le arrow functions).

Il valore della variabile `this` viene valutato a run-time.

```
let john = {
  name: "John",
  greet(){
    console.log(`Hi, I'm ${this.name}`);
  }
}

john.greet(); // Hi, I'm John!
```

Costruttori

Per creare più oggetti simili, si possono usare dei costruttori, che sono delle normali funzioni che (per convenzione) iniziano per lettera maiuscola e vengono invocati tramite la parola chiave "**new**".

Quando una funzione viene eseguita attraverso il new, in ordine:

1. Viene creato un nuovo oggetto ed assegnato a "this";
2. Viene eseguito il corpo della funzione (che solitamente modifica il this);
3. Viene ritornato il valore di this.

```
function Pet(name, species){  
  this.name = name;  
  this.species = species;  
  this.age = undefined;  
}  
  
let rick = new Pet("Richard", "Lizard");
```

Per evitare di ottenere errori quando si accede ad una proprietà indefinita di un oggetto, si può utilizzare l'operatore di **Optional Chaining (?.)**, che si ferma immediatamente se l'operando a sinistra è indefinito, ritornando undefined.

Le proprietà di un oggetto, oltre ad avere un valore, posseggono tre attributi speciali (**flags**), che di default sono settate a true:

- **Writable**: se true, consente di modificare il valore della proprietà;
- **Enumerable**: se true, consente di ciclare sulla proprietà;
- **Configurable**: se true, consente di cancellare la proprietà o di modificarne i flag.

Quando si accede ad una proprietà (in lettura o scrittura), vengono eseguite delle particolari proprietà (funzioni) dette **getter** e **setter**, che si dichiarano rispettivamente con le parole chiave get e set. Tramite i flags potrebbe essere utile, ad esempio, rendere accessibile soltanto un getter anziché le specifiche proprietà contenenti i valori.

Lezione 6

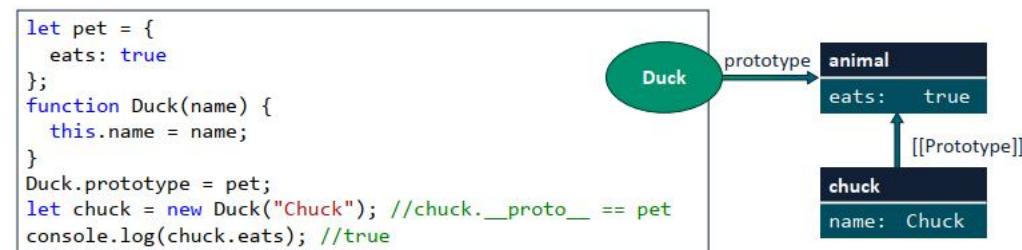
PROTOTIPI ED EREDITARIETÀ'

JavaScript implementa il concetto di ereditarietà attraverso l'uso della **prototipazione**: gli oggetti posseggono una speciale proprietà detta **[[Prototype]]**, che può assumere come valore "null" oppure puntare (tramite riferimento) ad un altro oggetto. Se cerchiamo una proprietà in un oggetto, ma questa non è presente, allora si cerca nella catena di prototype superiori. Si possono ottenere o modificare i valori di **[[Prototype]]** utilizzando i metodi **Object.getPrototypeOf()** e **Object.setPrototypeOf()**.

Le operazioni di cancellazione e scrittura lavorano direttamente sull'oggetto.

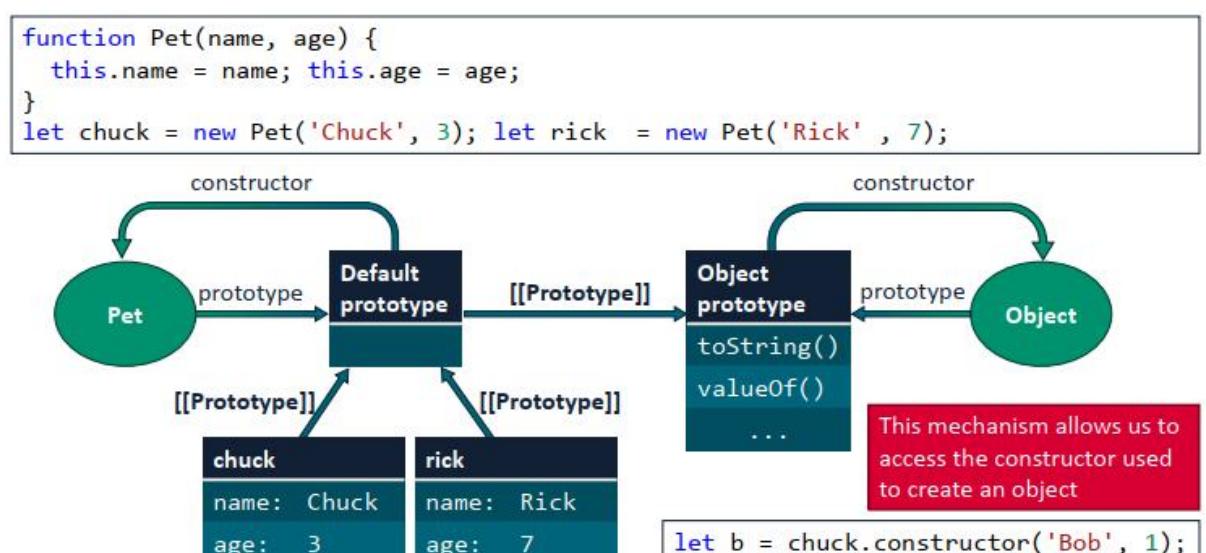
Quando la **proprietà prototype** di un costruttore è a sua volta un oggetto, utilizzando l'operatore new viene automaticamente settato il **[[Prototype]]** dell'oggetto appena creato.

Nota bene: Object.prototype è una semplice proprietà, mentre **[[Prototype]]** è una caratteristica del linguaggio che specifica il prototipo dell'oggetto.



È possibile aggiungere metodi ad oggetti già creati in due modi:

- modificando il costruttore, aggiungendo esplicitamente un nuovo metodo;
- aggiungendo, tramite dot notation, un metodo alla proprietà dell'oggetto (Object.prototype.method = method).



STRUTTURE DATI

ARRAY: memorizzano sequenze **ordinate** di valori. Si possono dichiarare tramite l'uso di parentesi quadra oppure con un costruttore di Array.

Gli indici iniziano da **0**, e la proprietà **length** contiene l'**indice massimo +1**, e non (come ci si potrebbe aspettare) il numero di valori nell'array. È possibile inserire dati di tipo diverso all'interno dello stesso array.

```
//array literal syntax  
let a = ["HTML", "CSS", "JS"];  
console.log(a); //Array(3) [ "HTML", "CSS", "JS" ]  
  
//constructor syntax  
let b = new Array("HTML", "CSS", "JS");  
console.log(b); //Array(3) [ "HTML", "CSS", "JS" ]
```

Gli oggetti Array posseggono metodi dedicati alla modifica degli elementi in essi:

- **push()**: aggiunge un elemento alla fine;
- **shift()**: rimuove un elemento dall'inizio e lo restituisce;
- **pop()**: rimuove un elemento dalla fine e lo restituisce;
- **unshift()**: aggiunge un elemento all'inizio;

Essendo oggetti iterabili, è possibile utilizzarvi dei costrutti iterativi come il **for...of** o il **forEach**.

Non è buona pratica utilizzare il **for...in**, in quanto è più lento e si rischierebbe di "perdersi" eventuali proprietà che non posseggono un valore definito.

```
let a = ["a", "b", "c"];
a[4] = "e";

for(let i = 0; i < a.length; i++)
  console.log(a[i]); //a,b,c,undefined,e

for(let item of a)
  console.log(item); //a,b,c,undefined,e

a.forEach( (value, index, array) => {
  console.log(`a[${index}]=${value}`);
});

for(let key in a){
  console.log(a[key]); //a,b,c,e
}
```

Si possono creare degli **array multidimensionali** definendo, come oggetti di un array, altri array.

Esiste una sintassi specifica per "spacchettare" degli array all'interno di una sequenza di caratteri, attraverso l'uso di "...rest".

```
let [a, b, ...rest] = [1,2,3,4,5];
console.log(a); //1
console.log(b); //2
console.log(rest); //[3, 4, 5]
```

ITERABLES: oggetti iterabili su cui è possibile utilizzare metodi come il **for...of**.

Per rendere un oggetto iterabile è necessario implementare un metodo speciale: **Symbol.iterator**. Questo metodo viene chiamato dal **for...of**, e ritorna un oggetto, sul quale è possibile accedere ad elementi consecutivi tramite il metodo **next()**, che ritorna oggetti della forma **{done: boolean, value: any}**, con **done** che è true solo sull'ultimo elemento dell'iterabile.

MAPS: insiemi iterabili di **coppie chiave-valore**, che a differenza degli oggetti normali, consentono di avere chiavi di ogni tipo.

Ad esempio:

`map.set(1, "Num");` e `map.set("1", "String")` creano due elementi diversi nella stessa map, mentre se avessimo usato un oggetto normale, il secondo statement avrebbe sovrascritto il primo valore.

<code>new Map()</code>	Creates the map
<code>map.set(key, value)</code>	Stores the value by the key
<code>map.get(key)</code>	Returns the value by the key, or undefined if not present
<code>map.has(key)</code>	Returns true if the key exists, false otherwise
<code>map.delete(key)</code>	Removes the key-value pair by the key
<code>map.clear()</code>	Removes everything from the map
<code>map.size</code>	Is the current element count

SETS: strutture dati che consentono di salvare valori a meno di ripetizioni.

<code>new Set([iterable])</code>	Creates the Set. If an iterable is provided, copies its values
<code>set.add(value)</code>	Stores the value, returns the set itself
<code>set.delete(value)</code>	Deletes value from Set. Returns true if value existed, false otherwise
<code>set.has(value)</code>	Returns true if value exists in the set, false otherwise
<code>set.clear()</code>	Removes everything from the set
<code>set.size</code>	Is the current element count

CLASSI

In JavaScript, le classi sono dei particolari tipi di funzioni.

Il costrutto "**class className { ... }**" fa, in ordine, le seguenti cose:

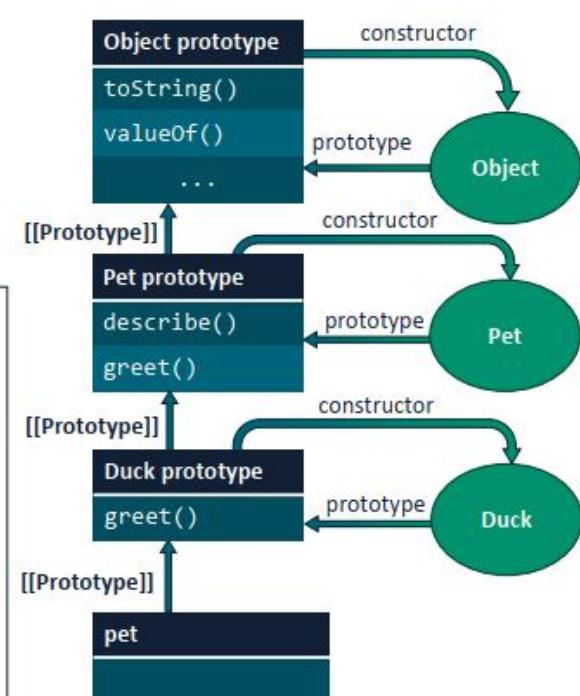
1. Crea un costruttore chiamato **className**, il cui codice è preso dal metodo **constructor()** (o è vuoto di default).
2. Salva gli altri metodi della classe in **className.prototype** (quindi nel prototipo del costruttore).

Anche le classi supportano proprietà, getter/setters, ed ereditarietà (attraverso l'uso della parola chiave **extends**, che è internamente implementata attraverso la proprietà **[[Prototype]]**).

```
class Pet {
  describe(){ console.log("I'm a pet"); }
  greet(){ console.log("..."); }
}

class Duck extends Pet {
  greet(){ console.log("Quack!"); }
}

let chuck = new Duck();
chuck.describe(); //I'm a pet
chuck.greet(); //Quack!
```



GESTIONE DEGLI ERRORI

Quando si incontra un errore, l'esecuzione dello script viene terminata immediatamente.

Per evitare che ciò accada si utilizza una sintassi **try/catch/finally**.

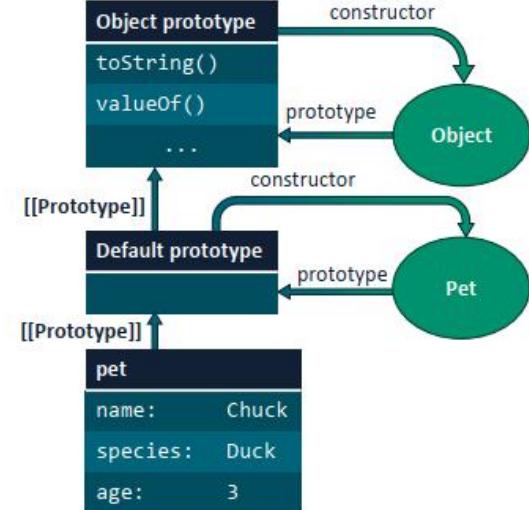
Deve esserci al più un solo blocco catch (all'interno del quale si possono gestire diversi errori), mentre i blocchi try e finally sono omissibili.

```
try {
    console.log("starting");
    x = 1; //forgot the let keyword
    console.log("assignment done"); // not executed
} catch (error) {
    console.log(error.name);
    console.log(error.message);
} finally {
    console.log("done");
}
```

L'operatore **instanceof** ritorna true quando la proprietà "prototype" del costruttore selezionato appare da qualche parte nella catena di prototipi dell'oggetto scelto per la verifica.

```
function Pet(name, species, age) {
    this.name = name;
    this.species = species;
    this.age = age;
}
const pet = new Pet('Chuck', 'Duck', 3);

console.log(pet instanceof Pet); //true
console.log(pet instanceof Object); //true
```



Gli **errori** in JavaScript si propagano verso l'alto fino a quando non vengono catturati o non raggiungono la radice dell'albero delle chiamate (interrompendo lo script). Possono essere lanciati con la parola chiave **throw**.

MODULI

JavaScript moderno consente di suddividere un programma complesso in parti indipendenti. Un **modulo** è un file .js, contenente classi e funzioni, creato per manutenibilità, riusabilità, e separazione degli interessi.

In particolare, vengono usate le parole chiave **export** (per etichettare parti di codice che devono essere accessibili al di fuori del modulo corrente) ed **import** (che consente di importare nel modulo corrente specifiche funzionalità dall'esterno).

Per importare un modulo nel proprio file, si utilizza la sintassi:

```
<script type="module" src="./script-module.js"></script>
```

Inoltre, per evitare eventuali problemi di concorrenza, attraverso i moduli è possibile selezionare con precisione cosa importare ed esportare utilizzando degli **alias**.

```
//pet-module.js
export function Pet(name, age){
    this.name=name; this.age=age;
}

let msg = "Hello"; //no need to export

export function greet(pet){
    console.log(`$ ${msg}, I'm ${pet.name}`);
}

//greet-module.js
let msg = "Howdy"; //no need to export

export function greet(name, message=msg) {
    console.log(`${message}, ${name}`);
}
```

```
//script-module.js, other modules are imported given their URLs
import {Pet, greet as greetPet} from './pet-module.js';
import {greet} from './greet-module.js';

let chuck = new Pet("Chuck", 7);

greetPet(chuck); //Hello, I'm Chuck

greet("Web Technologies"); //Howdy, Web Technologies
```

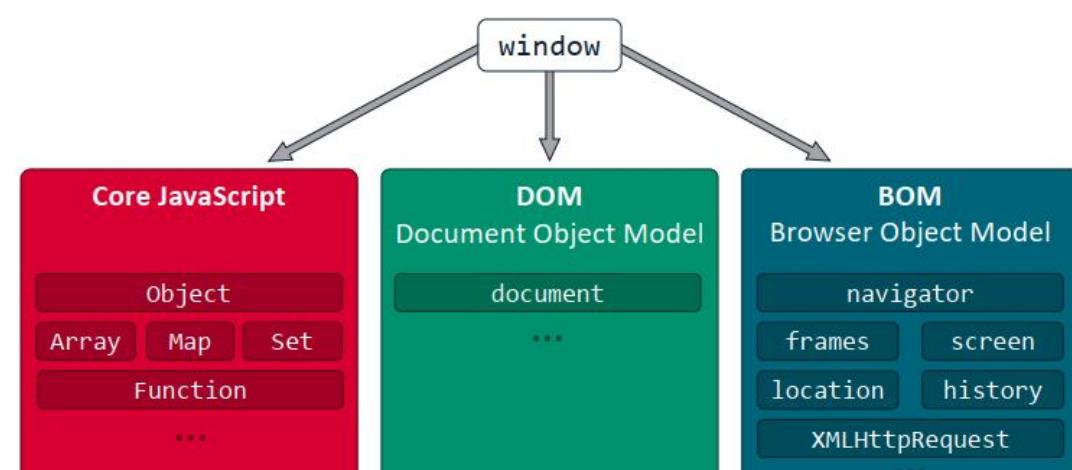
THE BROWSER ENVIRONMENT

Lezione 7

WINDOW: L'ambiente del Browser presenta un oggetto "radice" chiamato window, il quale:

- è un oggetto globale nel linguaggio JavaScript;
- rappresenta la finestra del browser, e consente di controllarla tramite codice.

Funzioni e variabili globali vengono rappresentate come proprietà dell'oggetto window.



Il **BOM** (Browser Object Model) fornisce contenuti (oggetti, metodi) aggiuntivi per dialogare con il browser in sé, e non con il contenuto dei documenti.

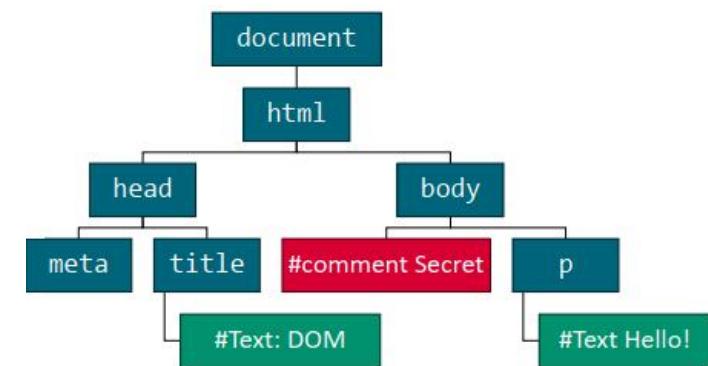
The Document Object Model (DOM)

Il DOM rappresenta il contenuto del documento corrente.

L'oggetto **document** è l'entry point principale della pagina web, e fornisce diversi metodi per accedere ai contenuti e per manipolarli.

Ogni **tag** HTML è un oggetto del DOM, come anche i **contenuti testuali**, gli **spazi**, le **newline** tra i tag ed i **commenti**, proprio come in un albero.

Gli **attributi** dei tag sono accessibili come proprietà dell'oggetto corrispondente al tag.



Il DOM identifica 12 tipi di nodi diversi, ma i più comuni sono quelli di tipo:

- **document** : rappresenta la radice (entry point) del DOM;
- **element** : rappresenta gli elementi HTML;
- **text** : rappresenta i contenuti testuali;
- **comment** : rappresenta i commenti.

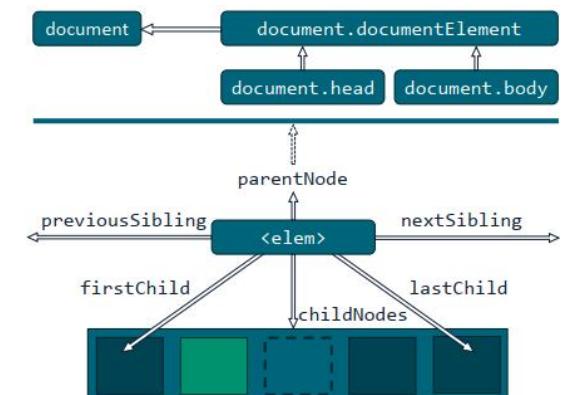
Il DOM fornisce diversi metodi per selezionare gli elementi del documento HTML:

- **document.querySelector(cssSelector)** : ritorna il primo elemento (se esiste) che fa match con il selettore passato in input, altrimenti restituisce null;
- **document.querySelectorAll(cssSelector)** : ritorna una lista di elementi (se ce ne sono) che fanno match con il selettore passato in input, altrimenti restituisce null;
- **document.getElementsBy*(parameters)** : ritorna una **HTMLCollection** di elementi che matchano con i parametri (ossia una "live collection", che viene automaticamente aggiornata e mostra sempre lo stato corrente del documento).

Se gli elementi che cerchiamo sono figli di un altro elemento, possiamo chiamare i suddetti metodi proprio sull'elemento padre così da **ottimizzare la ricerca**, che altrimenti avverrebbe nell'intero DOM.

I nodi del DOM contengono **riferimenti** (accessibili solo in lettura) ai loro genitori, ai fratelli, ed ai figli.

Method	Searches by...	Can be called on elements?	Returns Live Collection?
querySelector	CSS selector	✓	-
querySelectorAll	CSS selector	✓	-
getElementById	Id attribute	-	-
getElementsByName	Name attribute	-	✓
getElementsByTagName	Tag name or '*'	✓	✓
getElementsByClassName	Class attribute	✓	✓



Proprietà dei NODI

I nodi del DOM posseggono diverse proprietà a cui è possibile accedere:

- **nodeName/tagName** : per accedere (in sola lettura) alle informazioni sul tipo di nodo;
- **standard attributes** : per verificare il tipo degli attributi dell'oggetto HTML; queste sono anche **modificabili**.

Attraverso il DOM è possibile **creare** dei nuovi elementi HTML o **rimuoverne** degli esistenti, modificando la struttura del documento HTML, consentendo di creare pagine web **reactive**.

- **innerHTML** : per accedere al codice HTML contenuto in un **HTMLElement**;
- **outerHTML** : per accedere all'intero codice HTML di un **HTMLElement**;
- **document.createElement("tag")** : per creare un nuovo elemento **<tag>**;
- **document.createTextNode("text")** : per creare un nuovo nodo testuale;
- **node.append(element)** : per aggiungere element come ultimo figlio di node;
- **node.prepend(element)** : per aggiungere element come primo figlio di node;
- **node.before(element)** : per aggiungere element come fratello precedente di node;
- **node.after(element)** : per aggiungere element come fratello successivo di node;
- **node.replace()** : per sostituire node con element;
- **node.remove()** : per eliminare il nodo;

Attraverso il DOM è anche possibile **interagire con l'utente** che sta usando il browser web, in particolare:

- **alert("message")** : mostra un avviso testuale all'utente;
- **prompt("message")** : mostra un messaggio che invita l'utente ad inserire un testo come input;
- **confirm("message")** : mostra un messaggio che chiede all'utente di effettuare una scelta come input;

EVENTI

Gli eventi in un browser sono il **segnale che sia accaduto qualcosa**. Possono essere generati dal **comportamento degli utenti** (click, pressione di un tasto, movimento del mouse...), dallo **stato di un Form** (invio dei dati, selezione di una casella di testo...), o dal **documento** stesso (caricamento di elementi, richieste di rete completate...).

Per reagire agli eventi, esistono delle funzioni dette **handler**, che vengono eseguite al momento dello scoppio di un evento.

Si possono definire handler con l'attributo HTML **on<event>**, che ha come valore la funzione da eseguire, oppure in maniera dinamica è possibile, tramite JavaScript, aggiungere un metodo listener (**addEventListener**) all'elemento HTML che si vuole rendere reattivo.

All'interno di un handler, l'oggetto “**this**” viene valutato a run-time e si riferisce all'elemento che ha invocato l'handler.

Per gestire eventi più complessi si utilizza uno specifico **oggetto** atto a rappresentare l'intero **evento**. Eventi diversi hanno proprietà diverse.

Esiste in JavaScript il concetto di **Event Bubbling**: quando avviene un evento su un elemento, tutti gli handler relativi all'elemento vengono eseguiti, dopodiché vengono eseguiti anche gli handler per quell'evento sul padre dell'elemento, poi sui suoi altri antenati, fino ad arrivare al nodo radice del DOM. Questo meccanismo si applica a quasi tutti gli eventi, tranne alcune eccezioni come gli eventi di “focus”.

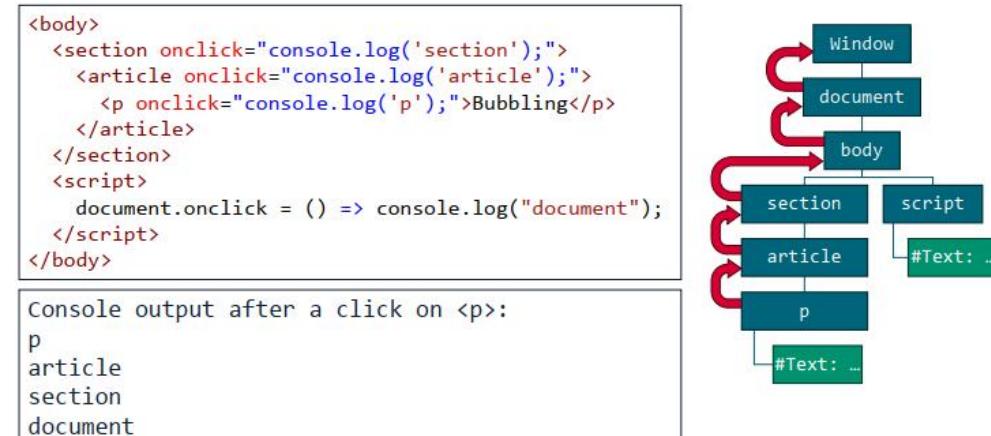
Questo concetto consente l'idea della **Event Delegation**: se molti elementi generano eventi simili, anziché creare tanti handler è possibile utilizzare un singolo handler su un antenato comune di questi elementi. La proprietà “**target**” consente di determinare quale sia stato tra i vari elementi quello a scaturire l'evento.

Per fermare il bubbling di un evento è possibile utilizzare il metodo **stopPropagation()** sull'oggetto evento.

Si possono anche **generare eventi** customizzati. Questi sono riconoscibili grazie alla proprietà “**isTrusted**” dell'evento, che a differenza di quanto avviene per gli eventi del linguaggio, è impostata a **false**. La generazione avviene tramite il metodo **dispatchEvent(eventName)**.

	click	User clicks (taps, for touchscreen devices) on element
Mouse	contextMenu	Mouse right-click on element
	mouseover/mouseout	Mouse cursor goes over / leaves an element
	mousedown/mouseup	The main mouse button is pressed / released over an element
	mousemove	The mouse cursor moves
Keyboard	keydownkeyup	A keyboard keys is pressed / released
Forms	submit	User submits a form
	focus	User focuses on an element (e.g.: <input>)
Document	DOMContentLoaded	The HTML has been parsed, DOM is fully built
Window	load	The web pages has been completely loaded and rendered

```
<input type="button" value="Click me" onclick="handleClick();">  
  
<script>  
  function handleClick(){ console.log("Click handled"); }  
  
  let input = document.querySelector("input");  
  input.onclick = () => {console.log("Tap");}; // overrides HTML-attrib. handler  
  
  input.addEventListener("click", handleClick); // adds new handler function  
</script>
```



ORDINE DI ESECUZIONE

Quando una pagina web viene caricata, **di default**, gli script sono presi ed eseguiti nell'ordine in cui appaiono, mentre il parse della pagina è in pausa.

È possibile aggiungere degli attributi alla dichiarazione di script esterni per modificare l'ordine di esecuzione di questi ultimi:

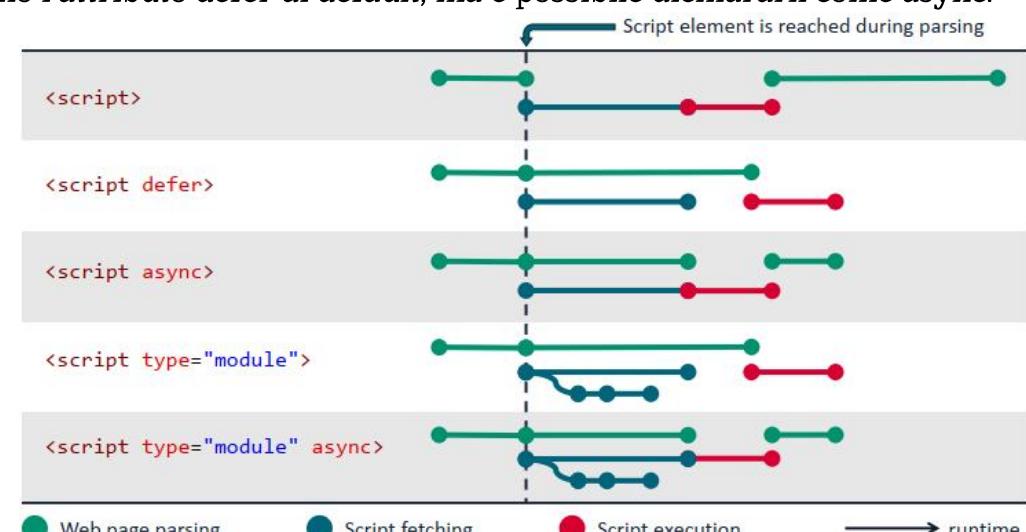
- **defer** : gli script esterni vengono scaricati in parallelo, ed eseguiti al termine del caricamento della pagina;
- **async** : gli script esterni vengono scaricati in parallelo, ed eseguiti non appena termina il download, anche prima che il parsing della pagina venga completato;

I **moduli** (che ricordiamo essere sempre script JS) hanno l'attributo defer di default, ma è possibile dichiararli come async.

L'ordine di esecuzione è molto importante perché potrebbe causare errori, come ad esempio l'accesso ad un campo del document che però non era stato ancora inizializzato.

In uno script asincrono, ad esempio, converrebbe non fare mai riferimento ad oggetti che interagiscono con il rendering della pagina.

Per una maggiore sicurezza, è possibile aggiungere apposite funzioni che controllino quando un evento possa essere gestito senza causare errori.



BROWSER STORAGE APIs

I web browser moderni forniscono molte API che possono essere usate tramite JavaScript per **salvare e recuperare informazioni**, come Cookies, Local/Session Storage, ed IndexedDB.

Cookies: Sono piccole stringhe testuali, parti esplicite del protocollo HTTP, utilizzate per trovare una soluzione al problema di HTTP di essere “stateless”. Vengono settati nelle risposte con l'header **Set-Cookie**, e sono salvati dai Browser per essere utilizzati (sempre tramite header) nelle successive richieste HTTP fatte allo stesso dominio.

L'oggetto document possiede una proprietà cookie, accessibile tramite **document.cookie**, che ha come valore una stringa di coppie nome=valore separate da “ ; ”, ognuna delle quali è un cookie.

```
if(document.cookie.indexOf("name=") === -1){ //no cookie called "name" found
    let name = prompt("Please, state your name:");
    document.cookie = `name=${name}; max-age=10` //expires in 10 seconds
}
let name = document.cookie.replace("name=", "");
document.querySelector("h1").innerHTML = `Hello, ${name}!`;
```

LocalStorage: consente di **salvare coppie chiave-valore** con diversi vantaggi: si possono salvare maggiori quantità di dati (rispetto ai cookie), si possono salvare soltanto stringhe, non bisogna lavorare con split o altri metodi per recuperare i dati dalle stringhe, ed esiste un oggetto localStorage diverso per ogni tripla **dominio/protocollo/porta**.

Una sua variante più leggera è **sessionStorage**, che a differenza della prima, esiste solamente all'interno di una tab del browser e non sopravvive ad eventuali chiusure della pagina o del browser.

```
<h1>Hello!</h1>
<script>
if(localStorage.getItem("name") === null){ //no "name" stored
    let name = prompt("Please, state your name:");
    localStorage.setItem("name", name);
}
let name = localStorage.name;
document
    .querySelector("h1")
    .innerHTML = `Hello, ${name}!`;
</script>
```

IndexedDB: si tratta di un DataBase costruito all'interno di un browser. Supporta tanti tipi diversi di dati ed ha più potenzialità di LocalStorage.

ASINCRONISMO

Di default, il codice **JavaScript** è eseguito in modo sincrono: ogni istruzione aspetta che la precedente sia completata.

Definiamo **Callback** delle funzioni passate come argomento ad altre funzioni, con l'assunzione che queste vengano chiamate nel momento opportuno.

Gestire tante callback annidate, ed eventualmente anche errori, potrebbe rendere il codice incomprensibile molto rapidamente, generando quella che si dice una **“Pyramid of Doom”**, ragion per cui in JavaScript moderno esistono le Promesse.

Le **Promesse** sono una sorta di collegamento tra codice **produttore** (ossia, che impiega del tempo per elaborare qualcosa) e codice **consumatore** (che utilizza i risultati dei codici produttori).

In JavaScript, le promesse sono degli **oggetti** che si creano, come tutti gli oggetti, attraverso dei costruttori. Il costruttore di una promessa prende come argomento un **executor**, ossia una funzione che viene invocata istantaneamente alla creazione della promessa e che prende in ingresso, a sua volta, **due callback**: resolve e reject.

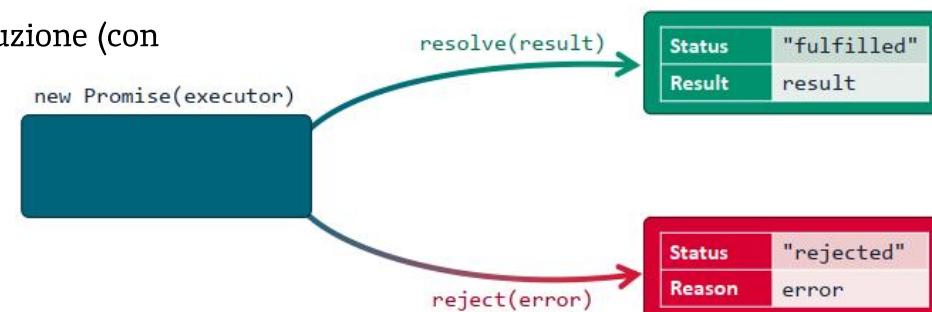
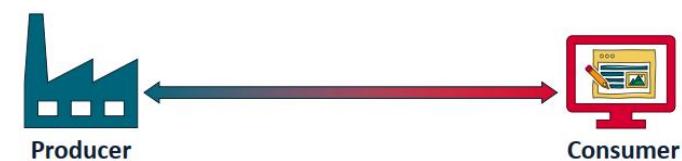
Quando il codice dell'executor termina, viene chiamata una delle due:

- **resolve(value)** : in caso l'esecuzione sia andata a buon fine ed abbia restituito un valore;
- **reject(error)** : in caso sia avvenuto un errore durante l'esecuzione (con error che è l'oggetto rappresentante l'errore avvenuto)

```
let promise = new Promise(function(resolve, reject){
    //Executor: producer code here
});
```

Callback Hell or
Pyramid of Doom!

```
function setupPage(){
    let result = 0;
    firstOperation(result, (r1) => {
        secondOperation(r1, (r2) => {
            thirdOperation(r2, (r3) => {
                console.log(`Result is ${r3}`);
            });
        });
    });
    console.log("Done");
}
```



Note: Status and Result/Reason are **internal properties** and can't be directly accessed!

Il codice consumatore riceve dunque la Promessa che i dati di cui ha bisogno arriveranno in futuro. Le azioni da compiere a seguito dell'adempimento (**fulfilled**) o del rifiuto (**rejected**) della promessa, possono essere specificati attraverso metodi specifici, tra cui il più importante è il **.then()**.

Il metodo **.then()** prende in ingresso due callback: una da eseguire se la promessa è andata abbon fine, ed una se questa è stata rifiutata. Eventualmente, si può specificare anche solo una delle due.

Promise possiede anche il metodo **.finally()**, che serve ad eseguire del codice indipendentemente dall'esito della promessa.

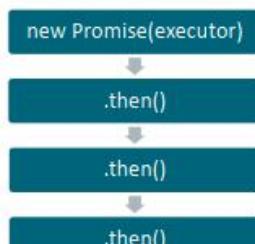
```
promise.then(
  function(result) { /* called when the promise is fulfilled */ },
  function(error) { /* called when the promise is rejected */ }
)
```

```
promise.then(
  (result) => { /* called when the promise is fulfilled */ }
)
```

```
promise.then(
  null,
  function(error) { /* called when the promise is rejected */ }
)
```

Grazie al **Promise Chaining** è possibile gestire sequenze di funzioni asincrone in modo elegante. Questo perché ogni **.then()** invocato ritorna sempre un nuovo oggetto Promise, dunque quando un handler restituisce un valore, questo non diventa altro che il risultato della nuova promessa.

```
promise //prints 1, then 2, then 4
  .then( (r) => {console.log(r); return r*2;})
  .then( (r) => {console.log(r); return r*2;})
  .then( (r) => {console.log(r); return r*2;});
```



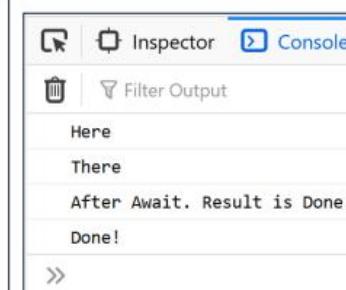
Gli oggetti Promise sono arricchiti da API che ne semplificano l'utilizzo:

- **Promise.all()** : prende in input un array di promesse, e ritorna una nuova promessa contenente tutti i risultati delle promesse iniziali. Se anche una sola delle promesse dovesse essere rifiutata, anche la **Promise.all()** ritornerebbe un rifiuto;
- **Promise.allSettled()** : meno stringente di **.all()**, in quanto ritorna un array di oggetti aventi come valore lo stato di tutte le promesse prese in input;
- **Promise.race()** : simile al **.all()**, con la differenza che viene ritornata solamente la prima promessa che termina l'esecuzione;
- **Promise.any()** : simile al **.race()**, con la differenza che aspetta la prima che venga effettivamente risolta, senza il rischio che si fermi ad una promessa rifiutata. In caso siano tutte rejected, **.any()** restituisce un array di messaggi di errore, uno per ogni promessa rifiutata.
- **Promise.resolve(result)** : crea una promessa già risolta, con risultato di ritorno "result";
- **Promise.reject(error)** : crea una promessa già rifiutata, con messaggio di errore "error";

Per facilitare il lavoro con le promesse, in JavaScript esistono due parole chiave molto utili:

- **async** : se posta prima di una funzione, assicura che questa ritorni sempre una Promise. Qualsiasi valore la funzione restituisca, questo verrà sempre racchiuso in un oggetto promessa. Stessa cosa per eventuali errori, avvolti in una promessa rejected.
- **await** : utilizzabile soltanto in funzioni dichiarate con **async**. Mette in pausa la funzione fin quando una determinata promessa non viene risolta.

```
console.log("Here");
async function f(){
  let promise = new Promise(function(res, rej){
    setTimeout(() => {res("Done!")}, 2000);
  });
  let result = await promise; //execution pauses here
  console.log(`After Await. Result is ${result}`);
  return result;
}
f().then(console.log);
console.log("There");
```



RICHIESTE NETWORK

JavaScript consente anche di recuperare dati da internet. In passato, era necessario usare oggetti XMLHttpRequest, i quali facevano utilizzo di callback per gestire lo stato degli eventi.

In JS moderno si può utilizzare il metodo **fetch()**, il quale prende in input un URL ed (eventualmente) un array di opzioni:

```
let promise = fetch("url", [options]);
```

Il metodo ritorna una promessa (che si risolve in un oggetto Response), e vengono usate opzioni per specificare il metodo HTTP, gli headers, ecc...

L'oggetto **Response** che ne deriva contiene diversi metodi per accedere al body in formati diversi (ma solo uno per risposta). Attenzione: il body potrebbe non essere disponibile subito: bisogna accedervi con cautela, e si può scegliere un solo metodo di lettura per ogni risposta.

.text()	Read the response body and return it as text
.json()	Parse the response body as JSON
.blob()	Return a Blob (Binary data with type)
.arrayBuffer()	Return an ArrayBuffer (low level repr. of binary data)
.formData()	Return a FormData object (represents data submitted via forms)

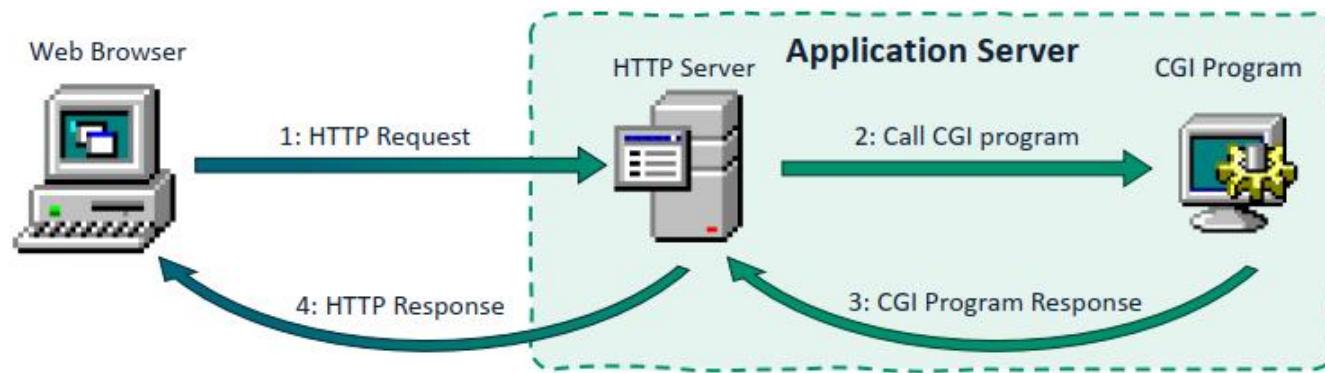
Pagine Web statiche

Il dinamismo che si può generare con JavaScript è comunque in un certo senso statico, in quanto esiste soltanto dentro al web browser. Quando un browser invia una richiesta ad una pagina web statica, la risposta ottenuta sarà sempre identica. Quanto fatto fin ora non ci permette di personalizzare pagine per utenti specifici o in base a richieste particolari, o semplicemente di creare risposte che varino l'una dall'altra.

Programmazione lato SERVER

Fin ora, i server web si limitavano a fornire file presi dalla document root. Dietro la programmazione lato server c'è, invece, l'idea che un server web possa **generare pagine "al volo"**, magari in base ai parametri ricevuti dalle richieste HTTP.

In passato, ciò avveniva grazie alla **Common Gateway Interface (CGI)**, ossia un'interfaccia che consentiva ai web server di eseguire programmi esterni per processare richieste HTTP. Il programma CGI chiamato legge i dati tramite variabili di ambiente e standard input, li processa, e fornisce i risultati mediante standard output.



SCRIPTING lato Server

Lo scripting lato server non è molto facile e veloce da effettuare manualmente, per questo esistono specifici linguaggi di programmazione e framework che semplificano le cose, tutti basati sull'idea di **mescolare codice HTML con codice lato server**, per poi passare il miscuglio ad un **interprete** che ne effettui il parsing per produrre del pure codice HTML:

- **PHP (PHP Hypertext Preprocessor)** - 1995
- **ASP (Active Server Pages)** - 1997 by Microsoft
- **JSP (Java Server Pages)** - 1999 by Sun

PHP

L'idea di base è la presenza di un interprete installato sul server che, nel momento in cui arriva una richiesta per un determinato file .php, viene invocato e processa tale file.

Per separare codice ed HTML si utilizzano dei **delimitatori** speciali:

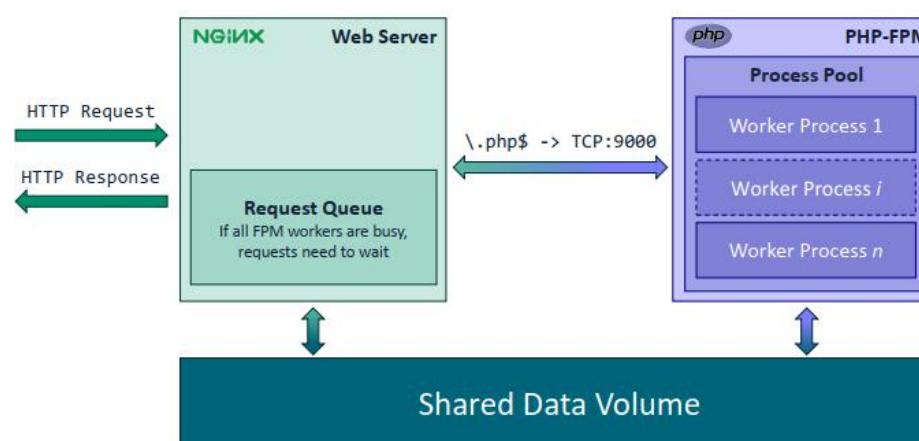
<?php ... code ... ?>

Le parti del documento che non sono tra i delimitatori speciali vengono lasciate intatte e messe in output, mentre tutto il codice compreso tra i delimitatori viene **interpretato**, e la sua traduzione viene inserita nel documento di output.

```
<!-- hello-world.php file -->
<!DOCTYPE html>
<html>
  <head>
    <title><?php echo "Hello PHP" ?></title>
  </head>
  <body>
    <h1>Hello PHP</h1>
    <?php $array = ["Hello", "PHP"] ?>
    <ul>
      <?php foreach($array as $item): ?>
        <li><?= $item ?></li>
      <?php endforeach; ?>
    </ul>
  </body>
</html>
```

Uno dei principali vantaggi di queste soluzioni fornite dallo scripting lato server è che, rispetto ad esempio a CGI, viene già sfoltita la lista di cose di cui occuparsi: ad esempio, le richieste vengono automaticamente pre-processate.

Architetture moderne per pagine web basate su PHP possono prevedere, ad esempio, l'utilizzo di **PHP-FPM** (FastCGI Process Manager), il quale consente, all'arrivo di una richiesta per un file .php, di processare tale richiesta per poi inviare il risultato al web server.



NODE.JS

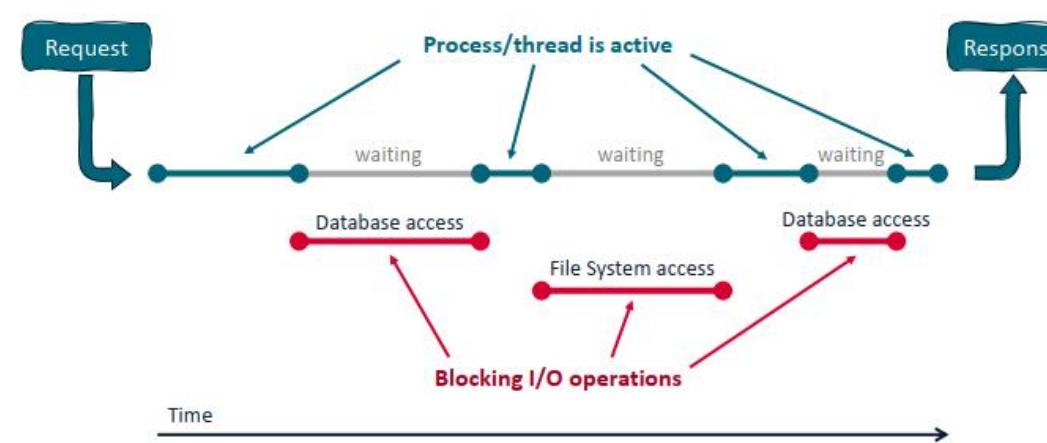
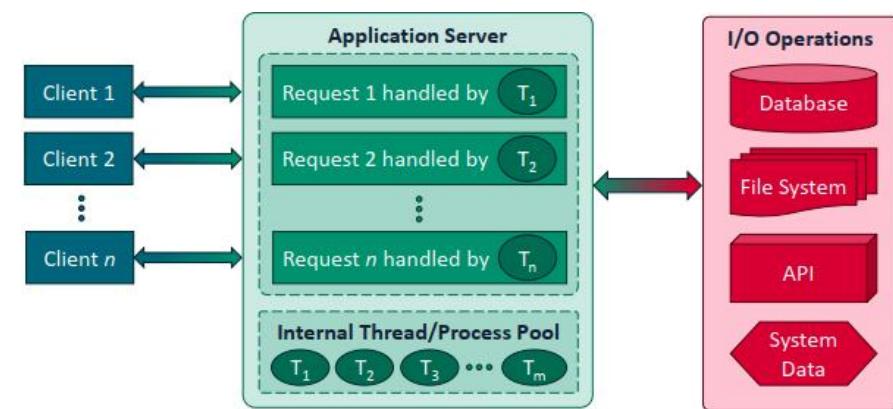
Node.js è un ambiente di esecuzione per codice JavaScript open-source e cross-platform. Segue un modello **basato su eventi, asincrono** di default, e con **input/output non bloccanti**.

Gestione multi-thread delle richieste

Soltanamente, negli ambienti di esecuzione, una richiesta è gestita da un solo thread/processo dedicato.

Questo, tuttavia, comporta che le **operazioni di I/O** possano diventare **bloccanti**: i threads/processi che devono gestire le richieste sprecano molto tempo in **attesa** del completamento di tali operazioni.

La scelta della gestione multi-threaded può dunque essere **inefficiente e poco scalabile**, in quanto non consente di gestire contemporaneamente più richieste di quante ne entrino all'interno del pool delle richieste.



Node.js, al contrario, è basato su una filosofia ben diversa: **un programma Node.js esegue un unico processo**. Non serve creare un nuovo thread per ogni processo che arriva, ma grazie ad un meccanismo detto "single-threaded event loop", possiamo eseguire un codice JS cercando di evitare blocchi nell'esecuzione.

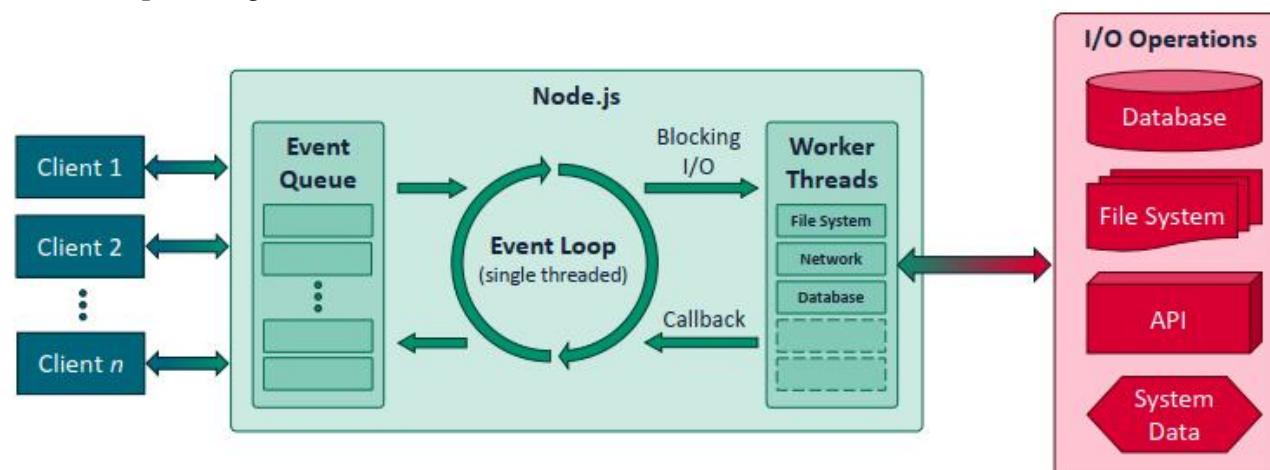
Nota bene: le operazioni sincrone dovrebbero essere eccezioni, utilizzate solo in caso non si possa farne a meno.

Il Single-Threaded Event Loop

Le richieste del client vengono salvate in una **coda degli eventi**, da cui l'**Event Loop** ne sceglie una per volta, iniziando ad eseguirla. Quando arriva una richiesta bloccante, la si assegna ad altri thread che lavorano **esclusivamente** su quella, mentre il processo dell'Event Loop continua il suo lavoro con la richiesta successiva.

Nel mentre, l'evento precedente viene soddisfatto e la richiesta viene inserita nuovamente nella coda.

Per ottimizzare il tutto è sempre meglio scrivere del codice non bloccante.



Alcuni comandi di Node.js:

- **nvs** : per installare Node;
- **node --version** : per visualizzare la versione installata;
- **node .\fileName.js** : per eseguire un file JavaScript;

È bene tenere a mente che l'ambiente di esecuzione in linea di comando è diverso da quello del browser: non c'è il DOM, non c'è una history, un localStorage. Tuttavia, in Node esiste una libreria standard che ci consente di lavorare col file system, di effettuare accessi alla rete, di modificare file, di implementare protocolli e molto altro ancora.

NPM: Node Package Manager

Driver che contiene milioni di pacchetti che è possibile importare nei propri progetti; in particolare, npm fornisce un modo per scaricare e gestire dipendenze per progetti Node.js.

Alcuni comandi di npm:

- **npm --version** : per visualizzare la versione installata;
- **npm init** : per creare un pacchetto npm nella directory corrente;
- **npm init es6** : per creare un modulo compatibile con EcmaScript;

Il comando init chiederà alcune informazioni generali sul progetto, dopodiché creerà un file **package.json** contenente tutte le informazioni riguardo al pacchetto, comprese eventuali **dipendenze** e **comandi** per eseguirlo.

Il file main viene eseguito quando il client importa il package. La proprietà **"scripts"** può essere usata per specificare dei **task** da riga di comando. Questi task possono essere eseguiti attraverso il comando **npm run taskName**.

Fa eccezione il task **start**, per cui basta **npm start**.

Per installare delle dipendenze si utilizza il comando:

npm install packageName

Così facendo, npm tiene traccia delle nuove dipendenze e le aggiunge nel file **package.json**. A questo punto, le dipendenze (del package richiesto e di sue eventuali dipendenze, in modo ricorsivo) vengono scaricate nella cartella **node_modules**.

Nota bene: quando si distribuisce un pacchetto npm, non serve aggiungere i moduli al versionamento: questi possono essere automaticamente installati tramite il comando **npm install**, che scarica tutte le dipendenze inserite nel **package.json**.

```
{  
  "name": "hello-web-technologies",  
  "version": "0.0.1",  
  "description": "Web Technologies' first npm package",  
  "main": "app.js",  
  "type": "module",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": [  
    "greetings"  
  ],  
  "author": "Luigi Libero Lucio Starace",  
  "license": "MIT"  
}
```

Quando effettuiamo delle modifiche al nostro codice in un progetto Node.js, abbiamo sempre bisogno di riavviare l'intero server per visualizzarle. Per ovviare a questo problema, esistono delle utility che tengono traccia del codice e di eventuali modifiche, e riavviano il server per noi quando necessario (**live reloading**).

Una di queste utility è **nodemon**, installabile tramite il comando: **npm install nodemon**.

Per usufruire di questa comodità, non serve far altro che avviare l'esecuzione del codice con il comando **nodemon app.js**.

Lezione 11

WEB APP con del semplice Node.js

Node.js ci consente di implementare server http molto semplicemente, sfruttando moduli built-in ed il single-threaded loop; inoltre, effettua pre-processing sulle richieste http e fornisce un'astrazione di richieste e risposte.

Tuttavia, ci sono comunque dei problemi di cui tener conto se vogliamo implementare una vera e propria web app:

- **Routing** : selezionare il codice da eseguire a seguito di una specifica richiesta;
- **Templating** : generare il file HTML della pagina;
- **Parsing** : effettuare il parsing del corpo delle richieste.

Routing

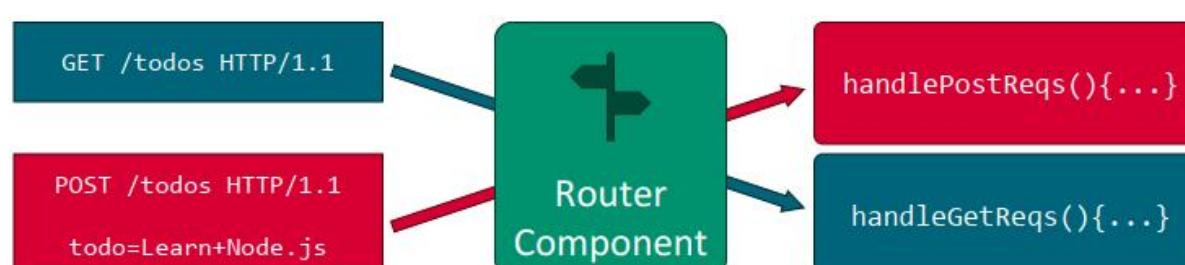
Procedura che stabilisce, all'arrivo di una richiesta, qual è il pezzo di codice da eseguire per gestirla.

C'è bisogno innanzitutto di indirizzare richieste diverse verso **path** diversi, ed ancora in maniera più specifica, potremmo aver bisogno di gestire richieste in maniera differente a seconda del **metodo** (GET, POST...) utilizzato per la richiesta HTTP.

Il componente Router può anche essere configurato affinché recuperi **file statici** dal file system, tramite ad esempio il metodo **fs.readFile()**.

Quando non c'è nessuna strada specificata per la richiesta in arrivo, si può **gestire l'errore** ritornando un **404** come risposta.

Nota bene: potrebbe essere necessario, in situazioni più complesse, che dei percorsi siano accessibili solo ad alcuni utenti o che dipendano dai parametri della richiesta, e ancora, che ci sia bisogno di più pezzi di codice per gestire la stessa richiesta.



Template Engines: PUG

Affinché il browser possa renderizzare il body di una risposta, le web app devono **produrre del codice HTML**. Nel caso in cui ci possa essere il rischio che si debbano ripetere più righe di codice uguali, o che le pagine siano troppo complesse, si ricorre all'utilizzo dei Template Engine, ossia degli **elaboratori di modelli**.

Pug utilizza una sintassi sensibile ad indentazione e spazi bianchi per scrivere templates che possano essere facilmente compilati in codice HTML. Per installarlo, basta eseguire il comando: **npm install pug**.

La funzione `pug.renderFile()` prende in input un template e restituisce in output l'HTML che si ottiene dall'esecuzione di tale template.

```
import pug from "pug"
//basic_example.pug is the file with the template to render
let html = pug.renderFile("./templates/basic_example.pug");
console.log(html); //html contains the generated HTML code
```

```
doctype html
html(lang="en")
  head
    title Hello Pug!
  body
    h1(class="foo") First Pug template
    p This is our first Pug template!
```

```
<!DOCTYPE html>
<html lang="en">
<head><title>Hello Pug!</title></head>
<body>
  <h1 class="foo">First Pug template</h1>
  <p>This is our first Pug template!</p>
</body>
</html>
```

Una grande comodità di Pug è che è possibile **riutilizzare template**. In particolare, dei template possono includerne altri attraverso proprio la parola chiave **include**.

Dato che spesso, all'interno di una pagina web, molti elementi si trovano sempre nelle stesse posizioni, è comodo avere una struttura di base da far ereditare a pagine più specifiche, le quali eventualmente possono effettuarne un override.

In Pug questa idea si risolve con l'**eredità dei template**: si utilizzano le parole chiave **block** (a definire parti di template che possono essere specializzate da eventuali discendenti, oltre che contenere qualcosa di default) ed **extends** (a specificare che un certo template ne estende un altro, eventualmente effettuando l'override di parte del template padre).

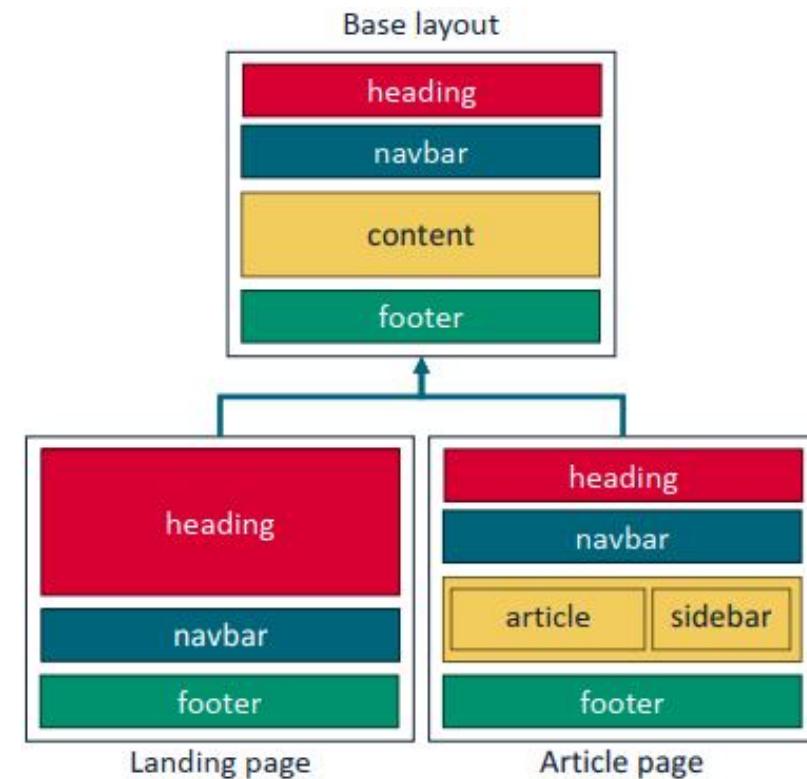
```
//- base-layout.pug
doctype html
html
  head
    title Pug Inheritance
  body
    block heading
      h1 Pug Inheritance
    block content
    block footer
      footer This is the default footer.

extends ./base-layout.pug

block content
  p The actual content of the homepage.

block footer
  footer This is a specialized footer.
```

```
<!DOCTYPE html>
<html>
<head>
  <title>Pug Inheritance </title>
</head>
<body>
  <h1>Pug Inheritance </h1>
  <p>
    The actual content of the
    homepage.
  </p>
  <footer>
    This is a specialized
    footer.
  </footer>
</body>
</html>
```



Al loro interno, oltre a poter definire variabili, i template consentono di passare ulteriori parametri, detti **locals**.

In particolare, è possibile passare questi oggetti (locals) come **parametro** delle funzioni `render()` o `renderFile()`.

Tramite la sintassi `#{}` è possibile poi accedere alle proprietà dell'oggetto passato: l'interprete valuta il contenuto, lo salva, e lo manda in output verso il codice HTML.

I template possono anche dover essere renderizzati in maniera differente a seconda di alcune condizioni, ragion per cui Pug supporta un costrutto **if/else**.

Allo stesso modo, è possibile anche **iterare** su sequenze di dati con costrutti come: **each item in items**.

Parsing del body delle richieste

Quando una richiesta viene processata, il body potrebbe non essere ancora disponibile.

Nel dettaglio l'**oggetto request**, per via del fatto che estende `stream.Readable`, genera due tipi di eventi diversi: **eventi data** (quando arriva una nuova parte di informazioni nel body della richiesta) ed **eventi end** (quando non ci sono più dati da prendere dallo stream di dati). Ciò significa che per leggere il body della richiesta c'è bisogno di operare il maniera **asincrona**, sfruttando le due tipologie di eventi generati.

Per creare una promessa bisogna passare una funzione executor nella quale inizializzare una variabile body ad un array vuoto. Dopodiché, tramite la `request` (passata come parametro), si possono specificare gli handler per i due eventi diversi.

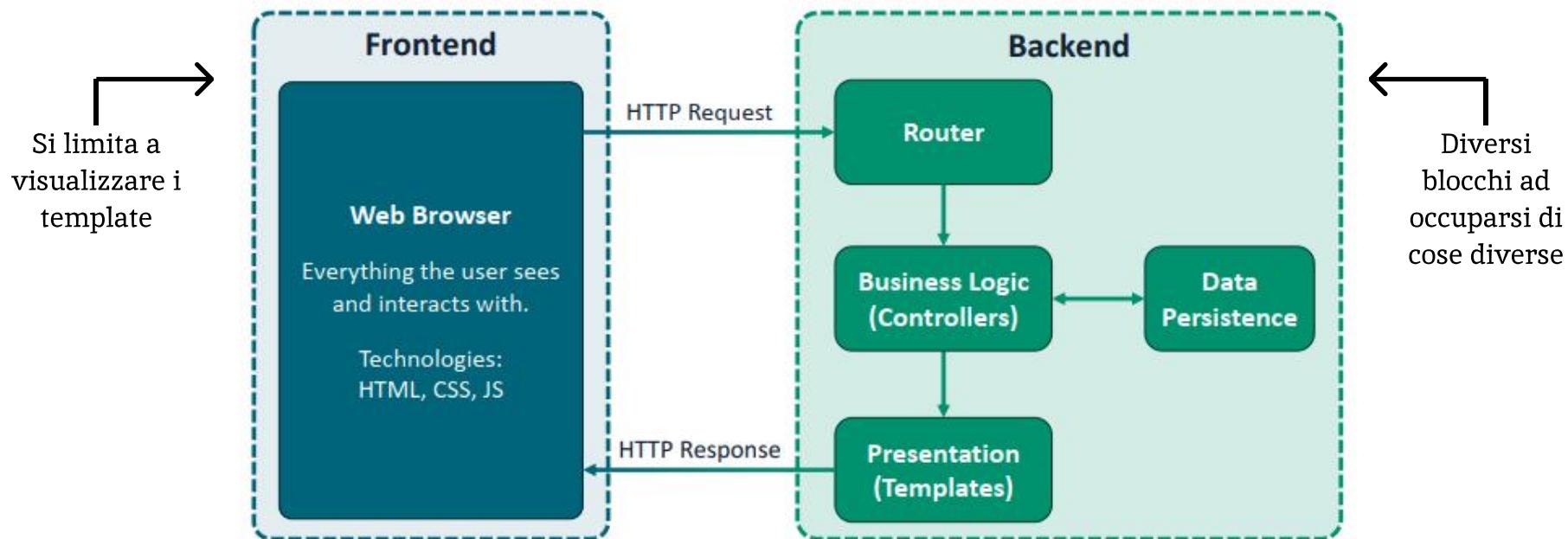
Più nello specifico, quando si verifica l'evento **data**, vengono pushate le nuove informazioni nel body, mentre quando si verifica l'evento **end**, si leggono tutti i dati che erano stati accumulati nel body, e se ne fa il parsing.

Alla fine si risolve la promessa passando i dati ottenuti.

```
function parseRequestBody(request){
  return new Promise((resolve, reject) => {
    let body = [];
    request
      .on('data', chunk => { body.push(chunk); })
      .on('end', () => {
        body = Buffer.concat(body).toString();
        // at this point, `body` has the entire request body stored as a string
        let data = parseRequestBodyString(body);
        resolve(data);
      });
  });
}
```

Una volta ottenuto il body sotto forma di stringa (**p₁=v₁&p₂=v₂&...**), non resta altro da fare se non effettuare degli **split** per ottenere nomi e valori dei parametri, ricordandosi anche di decodificare eventuali spazi e/o caratteri speciali.

ANATOMY OF A TYPICAL WEB APPLICATION



SESSION TRACKING

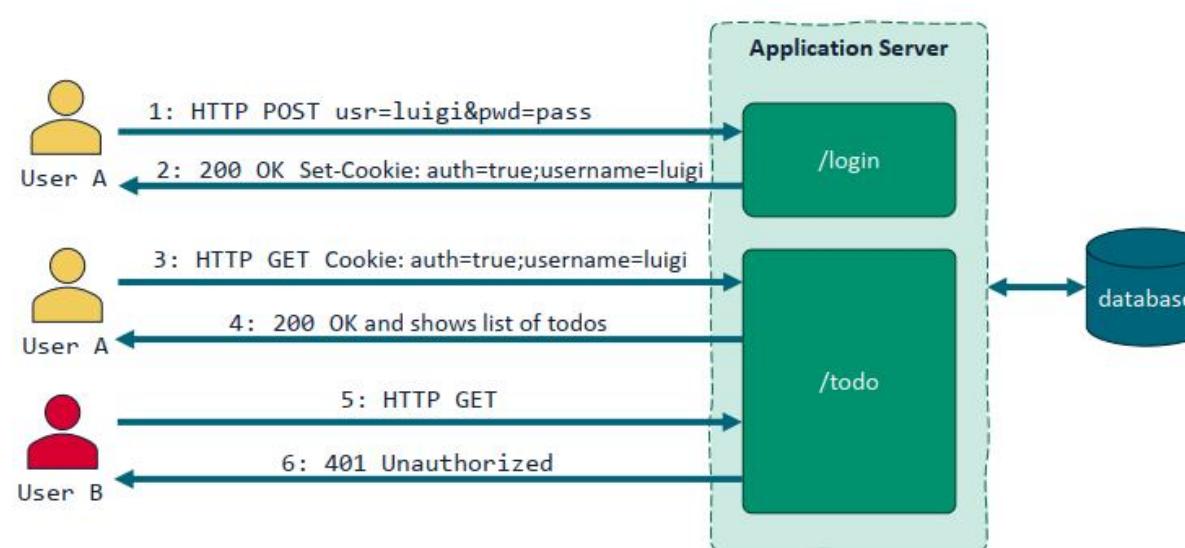
Lezione 12

Uno dei principali problemi di HTTP è che questo è **"stateless"**, ossia, una richiesta non contiene informazioni riguardo ad altre richieste passate. Il session tracking ha come obiettivo quello di superare tale mancanza del protocollo HTTP, permettendo al server di **mantenere informazioni sullo stato** in cui si trova il client in senso storico, su più richieste consecutive.

Il server potrà così riconoscere l'utente e capire se alcune operazioni siano già state effettuate o meno.

Cookies

Un modo molto semplice per effettuare session tracking attraverso le richieste HTTP è quello di usare i cookies: l'idea è quella che l'utente invii una richiesta ad una pagina dinamica, inviando alcuni parametri (come ad esempio username e password). La pagina web (o meglio, il server), a quel punto può settare dei Cookies nella risposta per tener traccia dell'interazione con tale client (**Set-Cookie: par1=value1;par2=value2;...**). Il client, in seguito, utilizzerà quegli stessi cookie per continuare a dialogare con la pagina web, "autenticandosi" per tenere traccia della sessione.

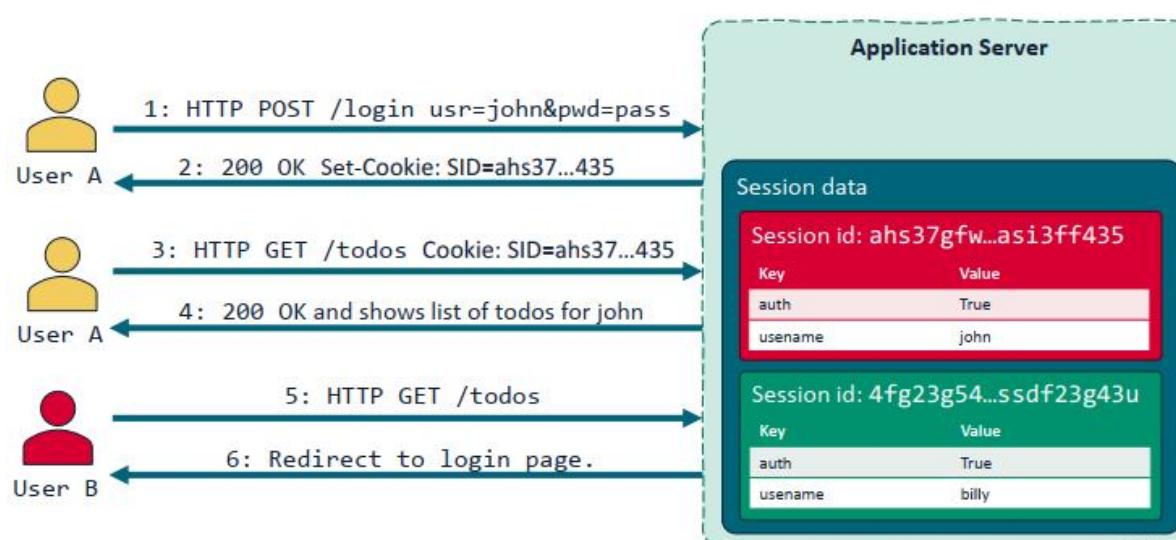


Come per quanto accade col **parsing** del body di una richiesta, anche in questo caso c'è bisogno di utilizzare dei metodi per identificare e leggere i cookies che arrivano. A differenza però di quanto accadeva col body, siamo sicuri che i campi header siano già definiti all'inizio del parsing: non c'è asincronismo. Un modo intuitivo per effettuare il parsing dei cookies è dividere la stringa ad ogni " ; ", e prendersi i campi di nome e valore di ogni cookie (rispettivamente, prima e dopo l'operatore " = "). Spesso c'è anche bisogno di effettuare URL Decoding.

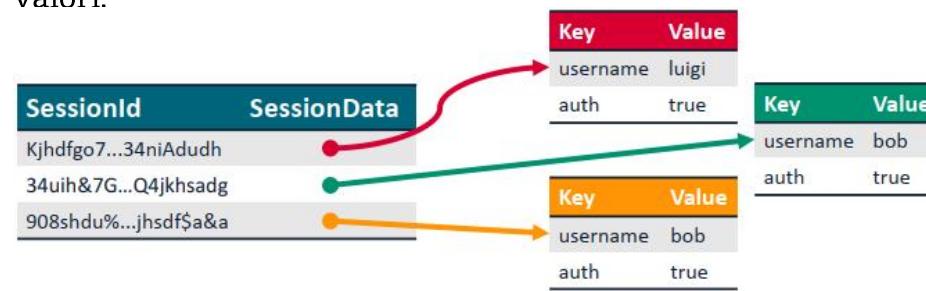
Il **problema principale** dell'approccio coi cookie è che questi possono essere **interamente controllati dal client**: un utente potrebbe manipolarli all'interno delle richieste, rendendo inutile un eventuale discorso di autenticazione. Si potrebbe risolvere utilizzando i **signed cookies** (cookie firmati), ossia dei cookie nascosti attraverso una chiave privata, ma ci sono anche altri problemi con questi approcci basati su cookie, come ad esempio quelli di performance: **la dimensione di un cookie è limitata** (generalmente, 4096 bytes), e gli host non possono salvare un grande **numero di cookie per ogni dominio** (generalmente, dai 20 ai 50).

Sessions

Per risolvere le mancanze dei cookies, si può utilizzare un **meccanismo di Web Session**: l'oggetto Session è una struttura dati simile ad una Map, fatta da **coppie chiave-valore**, specifica per ogni client. Le session **nascono e muoiono sul server**, ed essendo questo l'unico posto dove i dati sono conservati, non è possibile per i client (tramite browser) modificarne le informazioni. Ogni Session è identificata da un **id univoco** (session token), che solitamente viene passato al client tramite Cookie.



Attraverso Node.js è molto semplice creare un meccanismo di Web Session, in quanto è possibile utilizzare gli oggetti Map (che sono built in di JavaScript) facendo sì che, tramite una prima mappa, ogni Session Id venga associato ad una Session, e che questa sia a sua volta una mappa tra chiavi e valori.



```
class Session {
  constructor(){
    this.sessionStore = new Map();
  }

  createSession(){
    let sessionId = this.generateNewSessionId();
    this.sessionStore.set(sessionId, new Map());
    return sessionId;
  }

  storeSessionData(sessionId, key, value){
    return this.getSessionById(sessionId)?.set(key, value);
  }

  /* other methods ... */
}
```

Ci sono altri metodi, alternativi ai Cookie, per consentire lo **scambio del Session Id** tra client e server, come ad esempio:

- **URL Rewriting**: l'applicazione, quando genera una pagina, modifica ogni link interno a questa, concatenando agli URL il Session Id come **query parameter**;
- **Hidden form fields**: i Session Id vengono conservati all'interno di campi nascosti di un form. In particolare, deve esserci un campo nascosto (**type="hidden"**) che nel momento dell'invio del form da parte dell'utente, viene riempito col Session Id dal client e spedito al server.

Web Frameworks

Un framework, in generale, è un **insieme predefinito di strumenti software e convenzioni** che serve come base per sviluppare del software.

I framework Web nascono per aiutare in maniera specifica gli sviluppatori Web: forniscono un modo **strutturato** per organizzare applicazioni Web senza dover creare tutto da zero (gestione del routing, rendering dei templates, gestione delle web sessions, dell'autenticazione, parse delle richieste e dei cookies...).

Differenza tra Librerie e Framework

A differenza di quanto accade con le **librerie** (che sono semplicemente un insieme di funzioni che possiamo chiamare nel nostro codice per svolgere determinati compiti), utilizzando dei **framework** non è il codice dell'utente ad essere in controllo del flusso di esecuzione, bensì è il framework stesso a decidere cosa eseguire e quando farlo.

La caratteristica principale che distingue le librerie dai framework è quindi la **Inversion of Control** (IoC, inversione del controllo): è un principio secondo il quale non è più il programmatore (in maniera imperativa), ma è il framework che, in base a diversi fattori, sceglie quando eseguire il codice dell'utente. Tale principio viene anche detto Legge di Hollywood: “*Non chiamarci tu, ti faremo sapere noi, nel caso*”.



Componenti principali di un Framework Web

- **Funzionalità principali:** Routing; parsing delle richieste; validazione degli input; gestione dei Cookie e delle Sessioni.
- **Motore di Template:** aiuta nel rendering dinamico del contenuto dell'applicazione, separando il livello logico da quello di presentazione.
- **Meccanismo ORM (Object-Relational Mapping):** aiuta e semplifica l'interazione con il DataBase consentendo ai programmatore di lavorare su oggetti anziché su Query SQL.

Frameworks Opinionati

Un framework può essere più o meno “opinionato”: si dice **fortemente opinionato** quando non lascia molto margine di manovra al programmatore, in quanto è correlato da rigide convenzioni e decisioni prese dai creatori del framework; si dice, al contrario, **non opinionato**, quando consente al programmatore di fare tante cose ed in maniera diversa.

Il principale **vantaggio** dei framework fortemente opinionati è che questi consentono a progetti diversi di essere consistenti tra loro, velocizzandone lo sviluppo, poiché il tempo necessario alle scelte implementative è molto ridotto. Gli **svantaggi** invece riguardano l'alta curva di apprendimento per l'utilizzo del framework, e la poca flessibilità che potrebbe causare problemi in progetti specifici.

Express

Framework **veloce, non opinionato e minimalista** per Node.js.

Si aggiunge all'applicazione con il comando: **npm install express**.

Consente di creare applicazioni web che ascoltano richieste GET ed inviano risposte in pochissime righe di codice:

```
import express from "express";
const app = express(); // creates an express application
const PORT = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(PORT);
```

Routing: indica il modo con cui il framework determina come gestire richieste indirizzate ad uno specifico endpoint.

Una **route** si definisce con un metodo specifico dell'**oggetto app**:
app.metodoHttp(path, funzione callback)

```
app.post("/hello", (req, res) => {
  res.send("Hello Post");
});
```

Route Paths: potrebbero non necessariamente essere una stringa, ma anche, ad esempio, un'espressione regolare. In caso di overlap tra più route, quella applicata è la prima ad essere trovata.

Potrebbe essere utile in alcuni casi passare dei parametri all'interno del path, così da poterli utilizzare all'interno della route.

```
//if a get request is made to /student/42/exams/TecWeb
app.get("/student/:student_id/exams/:exam_name", (req, res) => {
  res.send(req.params); //{"student_id": "42", "exam_name": "TecWeb"};
});
```

Route Handlers: si possono specificare anche uno o più handler per le route. L'invocazione del metodo **next()** (che è una callback) specifica il successivo step nella pipeline delle funzioni (handler) da eseguire.

Per fattorizzare il codice, è possibile specificare il path della route in un solo punto all'inizio (**app.route()**) di una catena di handler, che vengono scritti a seguito, per evitare ridondanza.

```
let f1 = function(req, res, next) {console.log("f1"); next();}
let f2 = function(req, res, next) {console.log("f2"); next();}
let f3 = function(req, res, next) {console.log("f3"); next();}

app.get("/handlers", [f1, f2, f3], (req, res) => {
  res.send("Done with all handlers");
})
```

Router Modulari: col metodo **express.Router()** è possibile creare un oggetto **router** su cui definire delle route specifiche. Successivamente, tali oggetti possono essere importanti e “**montati**” all'interno di altre applicazioni Express.

```
//router.js
import express from "express";

export const router = express.Router();

router.get("/echo/:value", (req, res) => {
  res.send(req.params.value);
});
```

```
import { router as echoRouter } from "./router.js";
const app = express();
//other routes can be defined here as done earlier...
app.use("/custom", echoRouter); //load echoRouter (optionally in a certain path)
//requests to /custom/echo/:value will be handled by the router
app.listen(3000);
```

Metodi di Risposta: per evitare che la richiesta resti insoddisfatta, è necessario chiamare dei metodi appropriati sull'**oggetto risposta** (res).

Method	Description
<code>res.download()</code>	Prompt a file to be downloaded.
<code>res.end()</code>	End the response process.
<code>res.json()</code>	Send a JSON response.
<code>res.jsonp()</code>	Send a JSON response with JSONP support.
<code>res.redirect()</code>	Redirect a request.
<code>res.render()</code>	Render a view template.
<code>res.send()</code>	Send a response of various types.
<code>res.sendFile()</code>	Send a file as an octet stream.
<code>res.sendStatus()</code>	Set the response status code and send its string representation as the response body.

Rendering di un Template: notevolmente semplificato: se arriva una richiesta con path **/template**, viene renderizzato il file template che si trova in una cartella **views** di default.

Inoltre, i dati contenuti nell'**oggetto res.locals** sono accessibili a tutte le views.

```
const app = express();
app.set("view engine", "pug");

app.get('/template', (req, res) => {
  res.render("template");
});

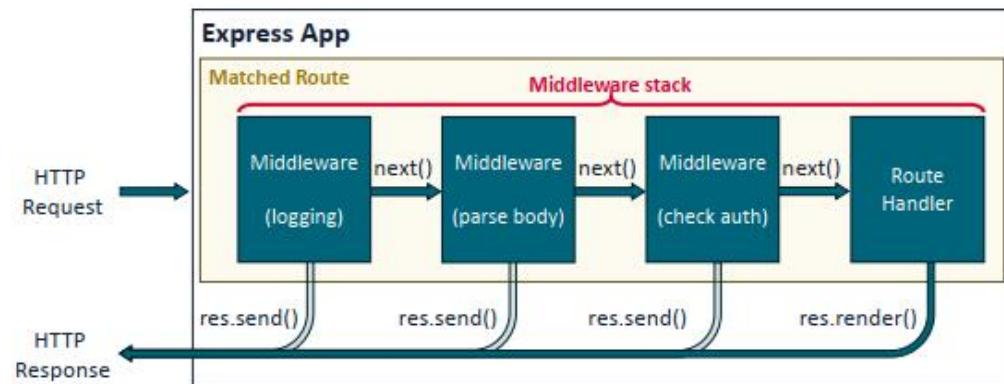
app.listen(3000);
```

```
//- file: /views/template.pug
doctype html
html
  head
    title Pug Template rendered in Express
  body
    h1 Rendering a Pug Template in Express
    p It's a straightforward process.
```

Il Ciclo Richiesta - Risposta

Il ciclo richiesta - risposta è alla base del framework Express, e consiste in 3 step:

1. Creazione degli oggetti richiesta e risposta;
2. Esecuzione di una sequenza di azioni (middlewares) per processare la richiesta (parsing del body, dei cookie, instaurazione di sessioni...);
3. Preparazione ed invio della risposta.



MIDDLEWARES

Sono delle **funzioni** (eseguite nello stesso ordine in cui sono dichiarate) che prendono in input 3 argomenti: l'oggetto richiesta (**req**), l'oggetto risposta (**res**), ed una callback (**next()**);

Vengono eseguite nel tempo che passa tra l'arrivo di una richiesta e l'invio della relativa risposta.

Se un middleware chiama la funzione **send()** anziché la **next()**, allora il ciclo richiesta - risposta si ferma e viene inviata la risposta.

Gestione degli ERRORI

In Express bisogna fare una distinzione: se gli errori vengono generati durante l'esecuzione di **codice sincrono**, allora il gestore degli errori li cattura **automaticamente**; se un errore viene generato all'interno di **codice asincrono**, invece, l'handler degli errori non se ne accorge e c'è bisogno di **passare l'errore** ad Express in maniera esplicita, come argomento del metodo **next()**. La gestione degli errori può ovviamente essere customizzata: gli **handler degli errori** sono dei middleware speciali, in quanto hanno un ulteriore argomento iniziale (**err**).

```
app.get('/user/:id', async (req, res, next) => {
  const user = await getUserById(req.params.id)
  res.send(user)
})
```

```
app.get("/error", (req, res) => {
  throw new Error("An error occurred");
});
```

```
app.get('/readfile', (req, res, next) => {
  fs.readFile('/file-does-not-exist', (err, data) => {
    if (err) {
      next(err) // Pass errors to Express.
    } else {
      res.send(data)
    }
  });
});
```

Middleware utili

- **express.static**: ogni volta che arriva una richiesta, viene eseguito, e controlla se bisogna prendere qualcosa da una cartella. In tal caso, si tratta di una richiesta per un file statico, e dunque la gestisce automaticamente.
- **express.urlencoded**: si prende cura di leggere il body delle richieste che arrivano, ne effettua il parsing, e rende disponibili in maniera automatica, all'interno di **req.body**, delle proprietà che rappresentano i valori che sono stati trovati col parsing.
- **express.json**: fa la stessa cosa di **urlencoded()**, ma legge all'interno del body di richieste sotto forma di file json. Salva le proprietà dello **json object** all'interno di **req.body**.
- **express-session**: si tratta di un middleware esterno rispetto ad Express: va scaricato da npm; nota bene: così come di default, non è ancora pronto per essere mandato in produzione, ma c'è bisogno di prepararlo. Questo middleware semplifica la gestione delle web session: consente ad esempio di definire un contatore per tener traccia di quante volte la pagina viene visualizzata da un determinato utente. Il middleware **crea la sessione, scrive un cookie nella risposta e rende l'oggetto session disponibile nella richiesta**, il tutto in maniera automatica.
- **cookie-parser**: middleware esterno ad Express che semplifica la gestione dei cookie. Vede le richieste HTTP che arrivano, controlla se è presente un header cookie, e ne effettua il parsing rendendo disponibili i cookie trovati all'interno dell'oggetto richiesta in **req.cookies**.
- **connect-flash**: middleware esterno ad Express. Rende disponibili diversi metodi che consentono di mostrare **Messaggi Flash** (ossia, dei messaggi "pop-up" utili per dare feedback all'utente o per reindirizzare altrove) all'interno della pagina. Dopo aver settato i messaggi flash, si effettua un redirect su una specifica pagina che mostra il messaggio specifico. Dopo essere stati utilizzati una volta, i flash-messages vengono consumati e non sono riutilizzabili.

Lezione 14

OBJECT-RELATIONAL MAPPING (ORM)

Quando abbiamo a che fare con una base di dati relazionale, nel software esistono **due rappresentazioni** distinte dei dati: gli oggetti che effettivamente **allociamo**, e poi quelli che "vivono" all'interno delle tabelle **della base di dati**.

Le due rappresentazioni sono molto diverse tra loro, e l'obiettivo è quello di mantenerle costantemente allineate: quando un oggetto viene creato, questo viene istanziato nello Heap, ma ovviamente c'è bisogno che lo stesso oggetto venga creato anche nel database affinché si possano salvare effettivamente dei dati.

Le **librerie ORM** (ossia, librerie che effettuano Object Relational Mapping) permettono agli sviluppatori di concentrarsi su delle astrazioni, anziché sul concetto di query SQL, statement, ecc.... In questo modo si aumenta la produttività (in quanto l'utente è solo tenuto a dire com'è fatto l'oggetto) e viene reso più semplice il passaggio da una base di dati relazionale ad un'altra.

La base di dati completa, in questo senso, diventa una sorta di **parametro** che può essere tranquillamente cambiato.

In maniera più concreta, una libreria ORM permette di definire in modo **dichiarativo** come gli oggetti del dominio sono fatti: proprietà, tipi, ecc; una volta dichiarato tutto, la libreria **crea e gestisce le tabelle** relazionali per conto proprio, **fornendo** anche delle **astrazioni** che consentono all'utente di effettuare **operazioni CRUD** sui dati. In aggiunta, forniscono anche **strumenti di caching** e strumenti per la gestione di transazioni.

SEQUELIZE

Sequelize è un ORM per Node.js che supporta i principali DBMS relazionali ed offre un'API basata su promesse.

Può essere installato tramite npm col comando **npm install sequelize**, dopodiché è necessario **installare anche i driver** per la base di dati relazionale che si intende utilizzare (es: npm install sqlite3).

```
import { Sequelize } from "sequelize";

//create connection
const database = new Sequelize("sqlite:mydb.sqlite");
//const database = new Sequelize('postgres://user:pass@example.com:5432/dbname')

try {
  await database.authenticate();
  console.log('Connection has been established successfully.');
} catch (error) {
  console.error('Unable to connect to the database:', error);
}
```

Modelli

Definiscono come sono fatti i dati. In Sequelize, questi non sono altro che delle **classi che estendono l'oggetto Model**. Si possono definire modelli o chiamando il metodo **define** su una connessione al database, oppure creando una classe che estende Model e chiamando il metodo **model.init()**.

Quando definiamo dei modelli, stiamo soltanto dicendo a Sequelize come sono fatti i dati con cui andremo a lavorare. In qualche modo, tuttavia, bisogna dirgli anche di **allineare lo schema** di un Database con il modello definito: lo si fa attraverso la **sincronizzazione**: si utilizza il metodo **model.sync(...)**, ossia una funzione asincrona che ritorna una promessa. Con la chiamata di questo metodo, Sequelize crea una tabella nel database chiamata col nome del modello (al plurale), a meno che non si voglia specificare un nome personalizzato.

Alla chiamata di `model.sync(...)`, Sequelize può avere 3 diversi comportamenti:

- `model.sync()`: se la tabella non esiste, viene creata, altrimenti non fa nulla (default);
- `model.sync({force: true})`: se la tabella esiste già, la cancella e ne crea una nuova;
- `model.sync({alter: true})`: se la tabella esiste già, controlla tutte le colonne e la modifica (se necessario) alterando le colonne affinché questa corrisponda al modello; questa chiamata non è distruttiva rispetto ai dati già esistenti, ma potrebbe fallire.

Per sincronizzare l'intero database, è possibile chiamare il metodo `sync()` sull'oggetto di connessione al database.

Si possono **creare delle entità** (effettive, all'interno del DB) in due modi diversi:

- utilizzando il modello come se fosse un costruttore, chiamando in seguito il metodo `save()` sul modello;
- usando il metodo `create()` sul modello.

```
let janet = new User({ //We can first create an instance of a Model...
  name: "Janet",
  age: 22
});

await janet.save(); //...and then save it to the database
console.log("Janet saved to database.");

let riff = await User.create({ //Or we can use the static method Model.create()
  name: "Riff Raff",
  age: 26
});
console.log(`Riff Raff saved to database with id ${riff.id}.`);
```

Con Sequelize è possibile creare **associazioni**:

- **uno ad uno**: `AhasOne(B)`;
- **molti ad uno**: `A.belongsTo(B)`;
- **uno a molti**: `AhasMany(B)`; (viene creata una proprietà all'interno di A sotto forma di array);
- **motli a molti**: `A.belongsToMany(B, { through: 'C' })`; (gli elementi di A appartengono a molti elementi di B attraverso la giunzione C, che conterrà le chiavi esterne sia di A che di B).

Creazione, Recupero e Modifica dei Dati

Quando si **crea** una tabella, Sequelize crea una colonna chiamata col nome del modello a cui si fa riferimento affiancato al nome della sua chiave primaria, in questo modo: **NomeModelloChiavePrimaria**.

Si possono anche creare istanze con associazioni in un solo passaggio:

```
let janet = await User.create({
  name: "Janet",
  age: 22,
  Todos: [
    {todo: "Learn Sequelize"},
    {todo: "Learn Express"}
  ],
  {
    include: [Todo]
  }
}); // code creating Riff Raff // and its Todo goes here
```

name	age	id	createdAt	updatedAt
Janet	22	1	2023-12-09 10:03:50	2023-12-09 10:03:50
Riff Raff	26	2	2023-12-09 10:03:50	2023-12-09 10:03:50

todo	done	id	createdAt	updatedAt	UserId
Learn Sequelize	0	1	2023-12-09 10:03:50	2023-12-09 10:03:50	1
Learn Express	0	2	2023-12-09 10:03:50	2023-12-09 10:03:50	1
Do the time warp again	1	3	2023-12-09 10:03:50	2023-12-09 10:03:50	2

Ci sono tanti modi per il **recupero** dei dati, che può avvenire selezionando intere tabelle o solo parte di queste.

Di default, le associazioni non vengono recuperate, ma è possibile includerle manualmente:

```
let todos = await Todo.findAll({include: [User]});
for(let item of todos)
  console.log(`${item.todo} - ${item.User?.name}`);
```

Si possono anche recuperare entità attraverso l'utilizzo della **chiave primaria**:

```
let t = await Todo.findByPk(3);
console.log("t:", t.id, t.todo, t.done, t.UserId);
```

```
let todos = await Todo.findAll();
for(let i of todos)
  console.log(`#${i.id}: ${i.todo}`);
```

```
import { Op } from "sequelize";

let todos = await Todo.findAll({
  where: {
    id: { [Op.gte]: 2 },
    todo: { [Op.like]: '%Express%' }
  }
});
```

Con il metodo `setPropertyValue(...)`, seguito dal `save()` è possibile modificare tabelle, e con il metodo `destroy()` si possono eliminare righe o intere tabelle.

```
let learnSequelize = await Todo.findByPk(1);
learnSequelize.setPropertyValue('done', true);
await learnSequelize.save(); //updates entity on db

let learnExpress = await Todo.findByPk(2);
await learnExpress.destroy(); //deletes entity from db
```

todo	done	id	createdAt	updatedAt	UserId
Learn Sequelize	0	1	2023-12-09 10:03:50	2023-12-09 10:03:50	1
Learn Express	0	2	2023-12-09 10:03:50	2023-12-09 10:03:50	1
Do the time warp again	1	3	2023-12-09 10:03:50	2023-12-09 10:03:50	2

todo	done	id	createdAt	updatedAt	UserId
Learn Sequelize	1	1	2023-12-09 10:03:50	2023-12-09 10:15:12	1
Do the time warp again	1	3	2023-12-09 10:03:50	2023-12-09 10:03:50	2

PRO e CONTRO degli ORM

PROS:

- **Astrazione:** si lavora con oggetti e modelli anziché con SQL;
- **Produttività:** meno codice duplicato, e schema generato in maniera automatica;
- **Portabilità:** facilità nel cambiare RDBMS;
- **Leggibilità e Consistenza:** più comprensibile dell'SQL, ed impone pattern e convenzioni.

CONS:

- **Difficoltà:** alta curva d'apprendimento;
- **Complessità:** particolari operazioni potrebbero risultare complesse con un ORM;
- **Vincoli:** potrebbe essere complicato passare da un ORM ad un altro;
- **Performance:** alcune operazioni potrebbero risultare più pesanti se svolte con un ORM, in particolar modo se eseguite in maniera non opportuna.

Struttura di una Applicazione Express

Express è un framework non opinionato: ci lascia un ampio margine di scelta nell'organizzazione delle applicazioni Web.

Ci sono alcune cartelle di default:

- **views:** contiene i template;
- **routes:** contiene la definizione dei Routers. Il codice qui presente è responsabile per l'indirizzamento delle richieste verso i controller appropriati;
- **controllers:** contiene la business logic per la gestione delle richieste;
- **models:** contiene la definizione dei modelli e delle connessioni al database;
- **middleware:** contiene middleware personalizzati;
- **public:** contiene file statici organizzati in cartelle.

Per la configurazione di una Web App si ricorre spesso all'utilizzo di informazioni sensibili (chiavi di API, password, credenziali...) e di impostazioni specifiche dell'ambiente di sviluppo (file di log, URL di database...): per evitare di scrivere queste informazioni in maniera hard-coded all'interno di vari file, queste andrebbero tutte salvate all'interno di un **file di configurazione** (configuration file).

I file di configurazione dovrebbero essere gli unici a contenere tali tipologie di informazioni e dovrebbero essere ignorati nel versionamento (tramite .gitignore, ad esempio... valutando l'idea di versionare solo una piccola versione di questi (dummy), senza informazioni sensibili, ed in cui l'utente potrebbe includere impostazioni personalizzate).

Per gestire le impostazioni si può utilizzare un file **.env** nella root del progetto.

Con Node.js si può utilizzare il pacchetto **dotenv** per estrarre il contenuto del file .env, lavorando su un oggetto **process.env**.

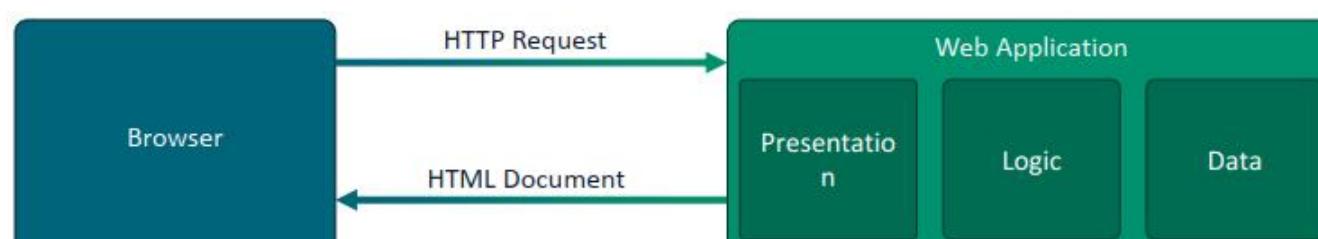


Web App tradizionali

Lezione 15

Nelle Web App tradizionali, la maggior parte delle operazioni viene svolta **lato server** (gestione dei dati, gestione della logica, gestione dell'interfaccia); questo perché le tecnologie per questa tipologia di applicazioni sono nate in un periodo in cui i Browser avevano capacità molto limitate.

Le Web App tradizionali non sono molto flessibili negli scenari moderni, in quanto software esterni che volessero accedervi, dovrebbero scaricare tutte le pagine HTML, farne il parsing, ed estrarre i dati. Peccano di **efficienza** (vengono trasferiti dati non sempre necessari) e di **robustezza** (qualsiasi modifica nelle pagine HTML potrebbe alterare le operazioni di recupero dei dati).



Il Trend Corrente

Oggigiorno si tende a dividere applicazioni web-based in almeno due componenti: un **backend** (responsabile per la logica, server-side), ed un **frontend** (responsabile per l'interfaccia grafica, client-side). Queste due componenti comunicano via Internet.

In altri termini, l'applicazione è in esecuzione sul server, ma ci sono anche dei software che girano sui vari client. Tra server e client, c'è Internet.



Per comunicare tra loro, i programmi (solitamente) utilizzano delle **API** (Application Programming Interfaces).

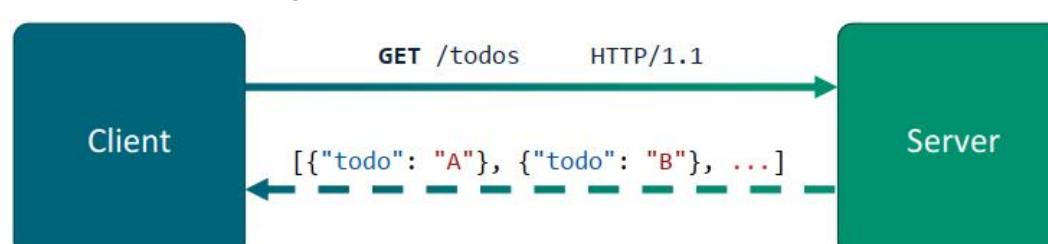
Per facilitare ed ottimizzare la comunicazione, le API fanno spesso uso del protocollo HTTP: si parla in questo caso di API Rest.

REST

REST non è uno standard, bensì si tratta di uno **schema architetturale** che specifica una serie di **principi** che definiscono un modo comune e consistente per far comunicare client e server utilizzando Internet (ed il protocollo HTTP).

REST si basa sul concetto di **risorsa**, ossia qualsiasi cosa sia così importante affinché qualcuno possa voler accedervi. Le risorse possono essere manipolate utilizzando i verbi del protocollo HTTP, e trovate tramite un **URI** univoco (che talvolta potrebbe corrispondere anche ad un path di risorse annidate, che però porterà ad una sola ed unica risorsa). Le API REST dovrebbero essere **stateless** per assicurare una maggiore scalabilità.

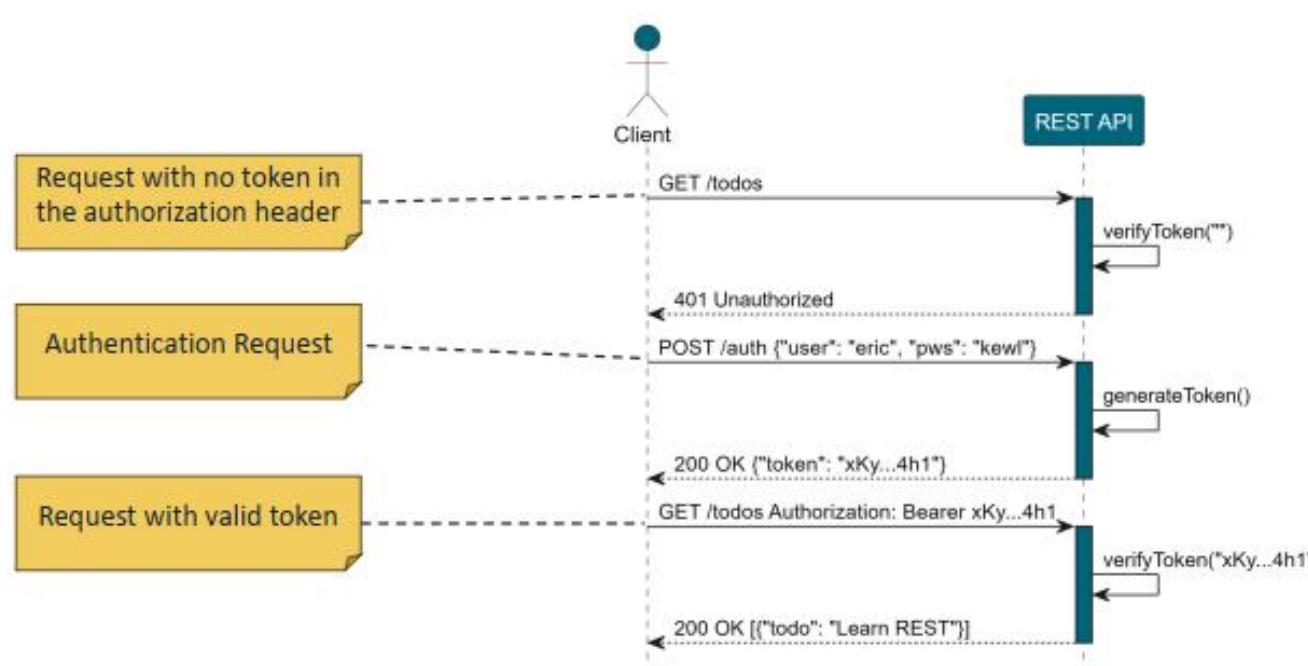
REST sta per **Representational State Transfer**: esistono lo stato dell'applicazione (che si trova sul client) e quello della risorsa (proveniente dal server), e questi due stati vengono trasferiti da una parte all'altra utilizzando rappresentazioni specifiche.



Sostanzialmente, per **implementare** una API REST non si deve far altro che mettersi in ascolto per delle richieste HTTP, e poi gestirle. Una volta gestita, si invia la risposta HTTP. Può sembrare non ci sia nulla di diverso rispetto alle web app tradizionali, tuttavia in questo caso, vengono utilizzate per i dati delle rappresentazioni più "machine-friendly" rispetto all'HTML, e non si è tenuti a fare affidamento sulle sessioni.

Dato che le API REST consentono agli utenti di manipolare risorse, c'è bisogno di un processo di **autenticazione** (solo gli utenti registrati possono accedere e modificare le risorse) ed **autorizzazione** (determinati utenti possono modificare solo determinate risorse, e non quelle di tutti).

Uno schema comune nelle applicazioni per l'autenticazione è quello dei Token: un po' come succedeva coi Cookie, il client manda una richiesta con username e password alle API, che (lato server) le validano e generano un token. Questo token (che è una particolare stringa) viene rispedito al client, il quale dovrà utilizzarlo in ogni successiva richiesta (inserendolo nell'header Authorization) per la quale è necessaria un'autenticazione. Le API, prima di inviare la risorsa richiesta, verificano la correttezza del token.

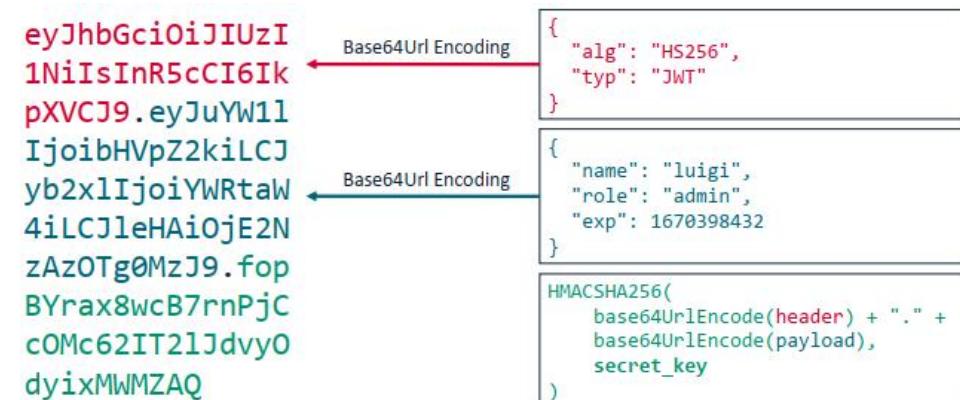


JSON WEB TOKEN (JWT)

JWT è uno standard che rende sicura la comunicazione tra due parti.

Si tratta di una stringa composta da 3 pezzi (separati da un punto): Header.Payload.Signature.

- **Header:** contiene informazioni sul tipo di token e sulla funzione crittografica utilizzata per la firma. È codificata tramite Base64Url Encoding, per cui facilmente invertibile;
- **Payload:** contiene una serie di affermazioni: possono essere qualunque cosa di cui vogliamo tener traccia. Esistono una serie di proprietà suggerite da usare, come exp (che indica la data di scadenza del token), ma in generale queste affermazioni sono personalizzabili. Anche questo è codificato in Base64 e dunque invertibile.
- **Signature:** firma che si ottiene applicando una funzione di hashing alla stringa ottenuta concatenando header, payload ed una chiave segreta nota soltanto al server che genera il token. Questa, ovviamente, non è invertibile.

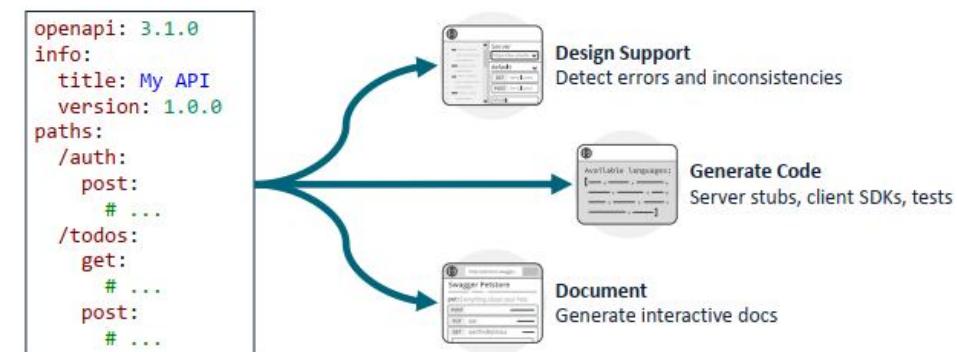


Le funzioni di **Hashing** mappano delle stringhe di lunghezza variabile in stringhe (dette hash o digest) di lunghezza fissa. Queste funzioni non sono invertibili ed è quasi impossibile generare collisioni.

OPENAPI

OpenAPI (o anche Swagger) è uno **standard formale per descrivere API REST** che fornisce numerosi vantaggi:

- **Standardizzazione:** garantisce pratiche di documentazione consistenti;
- **Documentazione:** può essere utilizzato per generare automaticamente una documentazione completa;
- **Generazione di codice:** consente di generare codice (e librerie) in maniera automatica, sia per client che per server;
- **Leggibile** sia da macchine che da essere umani;
- **Adottato** ampiamente come standard di settore.



L'idea alla base di OpenAPI è quella di avere in un **formato strutturato** tutte le informazioni riguardanti l'API, ossia specifiche, informazioni generali ed **endpoint** (percorsi).

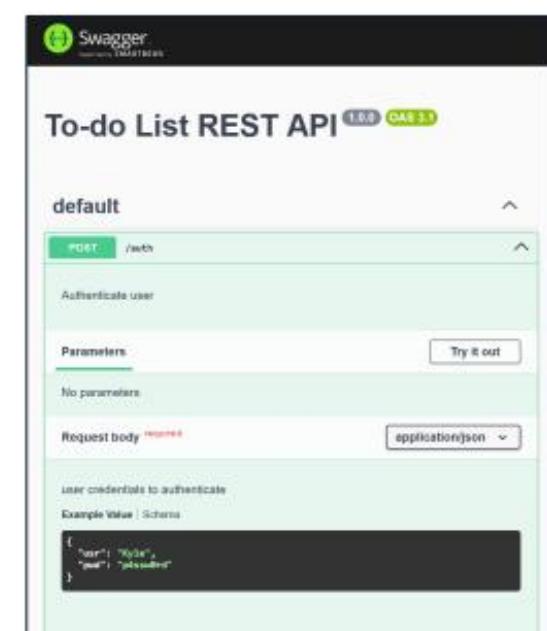
Per ogni endpoint di API si possono specificare determinate informazioni attraverso l'oggetto **Path**, come ad esempio i suoi metodi supportati, e le informazioni su tali metodi.

OpenAPI è supportato da numerosi **Tool** open source, tra i quali:

- **Swagger Editor:** editor che rende visibile specifiche e documentazione;
- **Swagger Codegen:** consente la generazione automatica di codice server-side e client-side (quest'ultimo, creando delle SDK per le nostre API, con la possibilità di scegliere tra diversi linguaggi);
- **Swagger UI:** interfaccia rapida ed intuitiva per la visualizzazione delle richieste HTTP.

Eventualmente, sarebbe possibile lavorare sulla **creazione di API** iniziando con lo scrivere le specifiche di queste, anziché il codice in sé: questo approccio si dice **OpenAPI-driven development**.

In particolare, tramite dei pacchetti Node, è possibile generare automaticamente sia delle specifiche OpenAPI a partire dalle annotazioni presenti nel codice sorgente (`npm install swagger-jsdoc`), sia delle documentazioni Swagger UI da Express (`npm install swagger-ui-express`), utilizzando un file di documentazione già esistente (magari proprio quello ottenuto da `swagger-jsdoc`).



TYPESCRIPT

JavaScript, essendo dinamicamente e debolmente tipato, è un linguaggio molto pratico per la manipolazione di semplici pagine web, ma rende difficile la vita dei programmatore nel momento in cui i programmi diventano più complessi, per via della difficile manutenibilità e della facilità nell'introdurre dei bug.

Per questo motivo, nascono i **TRANSPILER**, ossia dei linguaggi con molte utili funzionalità e che, compilati, vengono tradotti in semplice codice JavaScript; il più famoso di questi è TypeScript.

TypeScript può essere installato col comando **npm install -g typescript**, ed i file in questo linguaggio hanno estensione **.ts**.

Per l'esecuzione di file **.js** si utilizza il comando **tsc nomeFile.ts**.

Nota bene: codice JavaScript standard può essere benissimo eseguito tramite tsc, con la differenza che in caso di errori nel codice (come ad esempio, una variabile non definita), verrà dato un errore a tempo di compilazione. In questo senso, dunque, il vantaggio principale di TypeScript è la possibilità di avere controlli stringenti (come ad esempio quelli sui tipi delle variabili) sia a tempo di sviluppo che a tempo di compilazione).

Un file TypeScript eseguito con tsc viene **traspilato** in un file equivalente con estensione **.js**.

Il codice di un file TypeScript eseguito con tsc viene tradotto di default in una versione ES3 (1999) di JavaScript, a patto che non venga specificato un target diverso (es: **tsc -target es6 fileName.ts**).

Semantica del linguaggio

Dichiarazione di tipi: `let nomeVariabile: tipo = valore;`

Array: è possibile specificare tipi specifici per gli array, utilizzando una sintassi del tipo: `let nomeArray: type[] [values]`.

Any: è un tipo in TypeScript che rappresenta una sorta di wildcard; una variabile di tipo any può assumere qualsiasi valore, senza che il compilatore generi errori. Utilizzando questo tipo si sta in un certo senso ammettendo di non voler utilizzare i vantaggi di ts, in quanto si sta rinunciando a tutti i controlli sui tipi che questo offre.

Functions: TypeScript ci consente di specificare sia i tipi delle variabili di input che quelli dei valori di output delle funzioni.

Quando non dichiariamo esplicitamente il tipo di una variabile o di una funzione, il compilatore assume che questa sia di tipo any, dopodiché prova a capire il tipo specifico dal contesto, nel caso venga assegnato un valore di un certo tipo. Per non concedere questa "libertà" sui tipi, si può utilizzare il flag **-noImplicitAny** a tempo di compilazione.

```
function greet(name){ //function greet(name: any): string
  return `Hello ${name}`;
}

let arr = [1,2,3,4,5]; //let arr: number[]

let course = "Web Technologies"; //let course: string

greet(course);

course = {name: "Web Technologies", credits: 6};
//TS2322: Type '{name: string; credits: number;}' not assignable to type 'string'
```

Object: per la definizione di un oggetto c'è bisogno di listare le sue proprietà ed i corrispettivi tipi; nell'accedere agli oggetti, se si prova ad accedere a campi inesistenti o se si tenta di istanziare un oggetto con meno proprietà di quelle presenti nella definizione, si riceve un errore a tempo di compilazione.

È possibile tuttavia dichiarare **proprietà opzionali** con l'operatore `?` messo subito dopo il nome della proprietà.

Union: si ottengono combinando due o più tipi utilizzando una lista i cui elementi sono separati da "`|`"; il tipo unione, in altri termini, rappresenta valori che possono appartenere ad uno qualsiasi dei tipi di cui è stata fatta l'unione. Bisogna stare attenti, tuttavia, quando si applicano dei metodi sulle variabili di tipo union, in quanto alcuni metodi sono validi solo per un particolare tipo di dato, e potrebbero non esserlo per altri tipi contenuti nell'unione.

Aliases: per ottenere codice DRY (Don't Repeat Yourself, ossia, codice non duplicato) è possibile assegnare un nome ad un particolare tipo definito dall'utente tramite la parola chiave `type: type typeName = typeDefinition`.

Interfacce: sono un altro modo per assegnare un nome specifico agli oggetti. Utilizzando le interfacce, tuttavia, si perde un po' di libertà sull'utilizzo dei tipi, in quanto non possono esserci ambiguità, come poteva accadere con le Union. Questo accade per via della **tipizzazione strutturale** di TypeScript: controlla la struttura e le proprietà dei tipi soltamente quando deve andare a verificare la compatibilità di questi, a differenza di quanto accade ad esempio con Java, in cui la tipizzazione era nominativa e veniva guardato solo l'identificatore di un tipo ma non la sua struttura.

La **differenza principale tra type aliases ed interfacce** è che le seconde possono essere **estese**, mentre le prime non possono essere modificate una volta dichiarate.

Si possono in un certo senso estendere anche gli Alias utilizzando le **Inserction types**: attraverso l'uso del carattere `&` possono combinare le proprietà di ogni singolo tipo di cui si fa l'intersezione.

```
interface Pet {
  name: string
}

interface Pet {
  age: number
}

let matt: Pet = {name: "Matt", age: 2};
```

```
type Pet = {
  name: string
}

type Pet = { //TS2300: Dup. identifier
  age: number
}

let matt: Pet = {name: "Matt", age: 2};
```

Assertions: si può effettuare una sorta di casting, effettuando delle assertion nel momento in cui, da sviluppatore, conosco già il tipo che mi verrà ritornato da una funzione. Così facendo, è possibile avere accesso a quei metodi utilizzabili soltanto da tipi specifici.

Type literals: si possono definire dei tipi che possono assumere solo specifici valori (tramite le unions).

Enums: consentono allo sviluppatore di definire un insieme di costanti con un nome. Di default, ogni valore viene numerato da 0 in poi, nell'ordine in cui questi sono definiti (es: `enum Name { val1, val2, val3 }` assegna i valori 0, 1 e 2).

Funzioni

Si possono definire funzioni utilizzando le **function type expression**, ossia indicando un elenco di argomenti che la funzione deve avere in input, oltre che il tipo dell'elemento che produrrà la funzione. Queste utilizzano la stessa notazione delle arrow functions, assegnando un valore di ritorno: `type funcName = (arg1: type, arg3: type) => type;`

In questo modo è possibile anche definire funzioni che prendono delle callback come argomenti.

Nota bene: con questa notazione si possono anche definire funzioni che non prendono in input alcun argomento, ma che sono comunque assegnate ad una variabile, e non possono essere utilizzate con la sintassi delle funzioni vere e proprie.

```
let fn: () => string; //fn is a function that takes no args and returns a string

fn = () => {return "Hello Web Tech"};      //ok
fn = (x: string) => {return `Hello ${x}`} //TypeError!
```

Generics

Servono per scrivere del codice che sia **parametrico** rispetto al tipo: lo stesso blocco di codice può essere eseguito passando in input dei tipi diversi.

I generics sono ottimi per la riusabilità del codice.

Potrebbero sembrare simili al tipo any, ma c'è una differenza: quando si utilizza l'any, perdiamo informazioni sul tipo dell'oggetto quando questo ritorna, in quanto sarebbe sempre un oggetto di tipo any; utilizzando i genercис, invece, è possibile mantenere le informazioni sui tipi.

Il tipo sostituito nel nome contenuto tra le parentesi angolari, va ad essere sostituito ad ogni occorrenza della variabile.

```
function identity<Type>(x: Type): Type {
  return x;
}

let s: string = identity<string>("Hello"); //explicitly set Type to string
let n: number = identity(42); //let automatic type inference do its magic
let o: {title: string, artist: string} = identity({
  title: "Sultans of swing",
  artist: "Dire Straits"
});
```

Utilizzando i generics all'interno delle classi, possiamo scrivere il codice una sola volta ed utilizzarlo con più tipi diversi.

Altri tipi

void: tipo di ritorno per funzioni che non ritornano un valore;

unknown: tipo che rappresenta ogni possibile valore, però, mentre any dice al compilatore TypeScript di ignorare completamente quella variabile, questo è più conservativo: se il compilatore non sa su cosa sta lavorando, non consente di effettuarci alcuna operazione specifica;

never: tipo che rappresenta valori che non possono mai essere osservati; può essere utile nelle segnalazioni di errore.

Severità di TypeScript (strictness level)

Di default, TypeScript non è eccessivamente severo, infatti la dichiarazione dei tipi può essere opzionale, in quanto le variabili senza un tipo esplicito assumono il tipo any. Tuttavia, il livello di severità può aumentare utilizzando particolari flag a livello di compilazione (come ad esempio **-noImplicitAny**, oppure **-strictNullChecks**, che è disabilitato di default, e che forza a controllare, prima di ogni proprietà, che i loro valori non siano null o undefined).

```
@luigi → D/O/T/W/2/e/TypeScript $ tsc -target es6 .\strict.ts
@luigi → D/O/T/W/2/e/TypeScript $ tsc -target es6 -strictNullChecks .\strict.ts
strict.ts:13:13 - error TS18048: 'character' is possibly 'undefined'.
13 console.log(character.toUpperCase());
~~~~~
Found 1 error in strict.ts:13
```

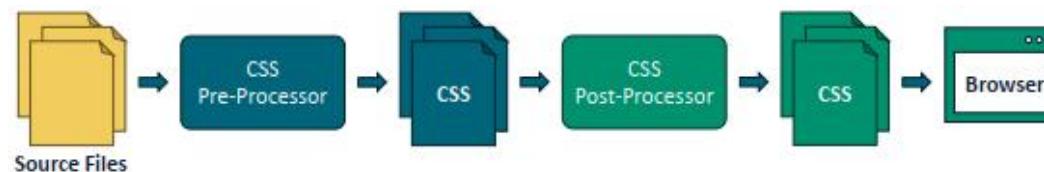
FRONTEND

Il codice frontend di applicazioni moderne può essere molto complesso, per via del grande numero di librerie e/o moduli utilizzati, del transpiling di codice, dell'ottimizzazione dei file, ecc.

Per questo motivo, spesso vengono utilizzati dei tool di supporto che vanno ad automatizzare determinate azioni, come ad esempio dei pre-/post-processori (come SASS).

CSS PRE-/POST-PROCESSORS

Quando le applicazioni sono molto grandi, i fogli CSS possono diventare enormi: esistono dei tool che servono ad ottimizzare i fogli di stile. L'idea dei pre e post processors è quella di scrivere del file sorgente, in un determinato linguaggio, che viene poi letto e processato da un Pre-Processor, il quale crea un file CSS. Questo file, viene letto a sua volta da un Post-Processor, che dà in output un altro file CSS ottimizzato.



I **Pre-Processors** si possono vedere come dei compilatori speciali, che prendono in input dei file specifici, ed i quali hanno una sintassi che può essere vista come una sorta di **evoluzione del CSS di base**, in quanto introducono nuove regole. In output, forniscono un foglio CSS.

I **Post-Processors** invece possono processare e trasformare fogli di stile già esistenti per eseguire, ad esempio, **downleveling** (per retro-compatibilità), **autoprefixing**, **ottimizzazioni** (come la minificazione o la cancellazione di codice inutilizzato), **individuazione di errori**...

SASS

Si tratta del pre-processore per CSS più utilizzato, e si autodefinisce "CSS coi superpoteri".

Può essere installato tramite npm con il comando: **npm install -g sass**.

Sass supporta due sintassi diverse: SCSS (sovra-insieme di CSS: c'è tutto quanto presente in CSS, ma comprende qualcosa in più) e la Sintassi Indentata (più vicina a Phyton, sensibile rispetto a spazi bianchi ed indentazione, e non fa utilizzo di graffe).

Variabili: sono dei modi di riutilizzare valori all'interno dei fogli di stile; quando un file Sass viene compilato, le variabili vengono sostituite dal loro valore.

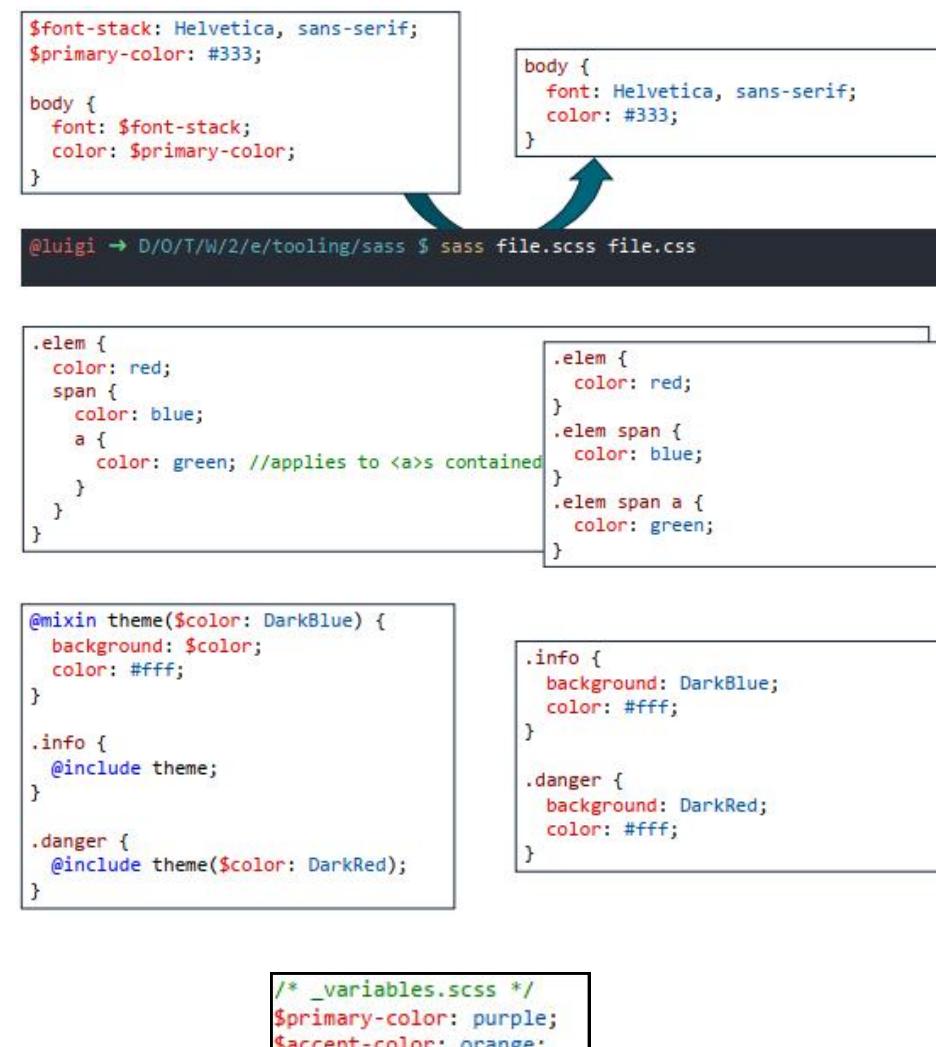
Nesting: Sass supporta il concetto di annidamento: consente di rendere il codice più leggibile; la versione compilata viene poi tradotta in CSS puro, senza annidamenti.

Mixins: il concetto di Mixin è quello di riutilizzare un gruppo di dichiarazioni in CSS / SASS. Dopo aver dichiarato un mixin (eventualmente, anche contenente parametri), si può includere dove ce n'è bisogno all'interno del foglio di stile. Si dichiarano ponendo la at-rule **@mixin** prima dell'identificativo, e si utilizzano tramite la regola **@include**.

Moduli: si possono creare dei moduli (i quali non sono altro che file SASS) da importare all'interno di altri file, tramite l'uso della regola **@use**.

Partials: si tratta di file SASS non stand-alone, ossia progettati con l'unico scopo di essere inclusi in altri file. Sass genererà dei file CSS da questi solo nel momento in cui serviranno effettivamente. Il loro nome inizia per **_**, e possono essere importati, come i moduli, tramite **@use**.

List e Map: utili strutture dati che possono servire nel momento in cui ci si trova dinanzi a diversi break-point, come ad esempio il resize di una pagina in base alle dimensioni dello schermo.



```
$list: (120px, 240px, 480px, 800px);
$list: (xs: 480px, md: 768px, lg: 1024px);
```

Flussi di controllo

In SASS ci sono anche modi per gestire quali sono gli stili da chiamare ed in base a cosa. In particolare:

- **@if/@else**: sono usati per emettere stili solo in base a delle condizioni;
- **@each**: è usato per iterare su tutti gli elementi di una lista/mappa;
- **@for e @while**: sono usati per iterare in base ad una condizione.

```
$sizes: ("xs": 480px, "md": 768px, "lg": 1024px);  
@each $sizeName, $sizeMaxWidth in $sizes {  
  @media screen and (max-width: $sizeMaxWidth) {  
    .hidden-$sizeName {  
      display: none;  
    }  
  }  
}  
  
@media screen and (max-width: 480px) {  
  .hidden-xs {  
    display: none;  
  }  
}  
@media screen and (max-width: 768px) {  
  .hidden-md {  
    display: none;  
  }  
}  
/* .hidden-lg omitted for the sake of brevity */
```

Frameworks CSS

Anche quando scriviamo CSS per il nostro frontend, ci troviamo spesso a dover risolvere gli stessi problemi per ogni progetto. I framework CSS sono un insieme di fogli di stile già fatti, che possono essere usati così come sono, o che (utilizzando strumenti di pre-processing) possono essere personalizzati.

Tutto ciò che bisogna fare è importarli all'interno delle pagine di un'applicazione, scaricandoli, o importandoli da una CDN (distribuzione di contenuti) remota.

Uno dei framework CSS più famosi è **Bootstrap**.

I file Sass di bootstrap possono essere importati ed è possibile effettuare l'override delle variabili al loro interno, così da personalizzare l'intero file o solo alcuni dei suoi moduli.

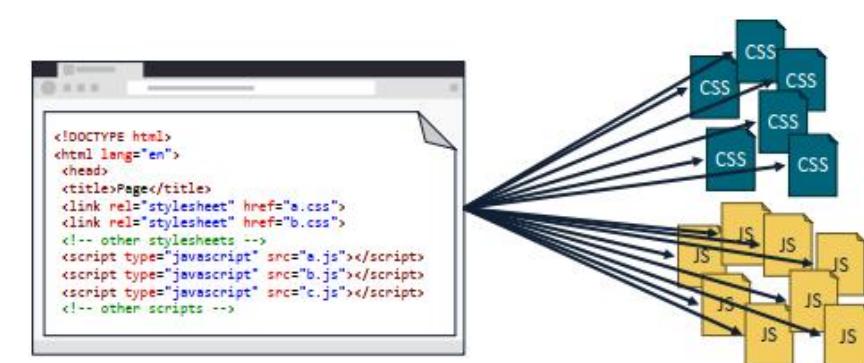
Dopo la compilazione tramite Sass, si ottiene un foglio di stile personalizzato.

```
// overriding Bootstrap variables  
  
$primary: #00647a;  
$secondary: rgb(254, 203, 110);  
  
@import "../node_modules/bootstrap/scss/bootstrap.scss";
```

BUNDLING

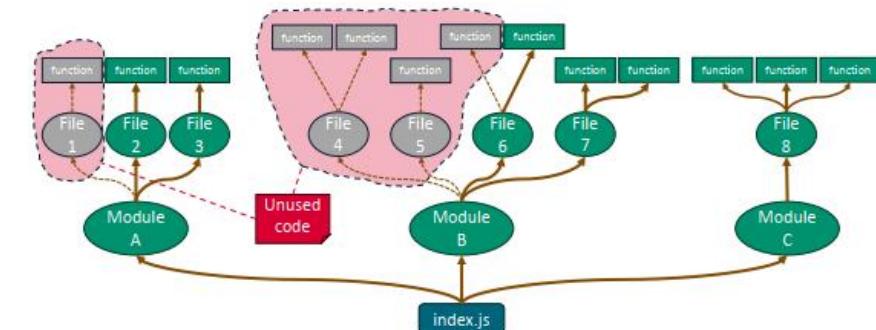
Consiste nel **combinare più file** CSS o JS in un singolo file per ridurre il numero di richieste HTTP effettuate nel caricare una pagina, aumentando la velocità dell'applicazione.

Così come è importante **l'ordine** in cui vengono definite e scritte le cose all'interno dei file HTML, JS e CSS, allo stesso modo è importante tener conto dell'ordine con il quale vengono inseriti i vari file all'interno di un Bundle.



TREE SHAKING

Si tratta dell'**eliminazione di codice morto**; a volte importiamo intere librerie nella nostra applicazione anche quando abbiamo bisogno solo di una piccola parte di queste. Con il Tree Shaking viene effettuata un'analisi degli import e degli export per capire quali parti possono tranquillamente essere "buttate via".



MINIFICATION

Lo step di "minificazione" consiste nell'**ottimizzare il trasferimento di file** rimuovendo spazi, indentazioni (in quanto le macchine non ne hanno bisogno per ragioni di leggibilità), modificando variabili (rendendole più corte, ecc...). Il file risultante viene detto "minificato", ed ha estensione **.min.js** oppure **.min.css** a seconda del file originario.

```
let msg = "Hello";  
let name = "Web Technologies";  
let str = `${msg}, ${name}!`;  
let b = true;  
if(b === true)  
  document.getElementById('msg').innerHTML = str;  
  
• 188 Bytes
```

↓

```
let msg="Hello",name="Web  
Technologies",str=`${msg},${name}!`,b=!0;!0==b&&  
(document.getElementById("msg").innerHTML=str);  
  
• 123 Bytes  
• 34,57% compression  
• Saved 65 Bytes
```

FILENAME FINGERPRINTING

Si tratta di un'operazione di Hashing del contenuto effettivo del file: così facendo, ad ogni minima modifica, c'è bisogno che venga generato un nuovo nome per tale file, oppure si potrebbero generare dei problemi a seguito del problema di non aver alcun "mantenimento dello stato" nella cache.

index-v5kLGx6_.css, index-DiAK_ebR.js

VITE

È un tool che **supporta gli sviluppatori** sia nella fase di sviluppo che di build dei frontend, ed è fatto da due componenti principali: un **server di sviluppo**, ed un **tool di build** che è possibile configurare per arrivare alla creazione di asset statici tra file js e css.

Si può utilizzare in un ambiente di esecuzione Web con il comando: **npm create vite@latest**.

Dopo l'utilizzo di questo comando, Vite genera per noi un ambiente di sviluppo con diversi tool utili, come ad esempio il live reload. In particolare, vengono introdotti degli script di default:

- **dev**: avvia il server, ed osserva eventuali cambiamenti tra i file per riavviare, in maniera automatica (quando necessario), l'applicazione;
- **build**: genera un bundle ottimizzato, pronto per essere inviato in produzione;
- **preview**: prende il bundle generato, e lo mostra con il server integrato di Vite.

```
{  
  "name": "hello-vite",  
  "private": true,  
  "version": "0.0.0",  
  "type": "module",  
  "scripts": {  
    "dev": "vite",  
    "build": "tsc && vite build",  
    "preview": "vite preview"  
  },  
  "devDependencies": {  
    "typescript": "^5.2.2",  
    "vite": "^5.0.8"  
  }  
}
```

Durante la fase di build, Vite fa diverse cose per noi:

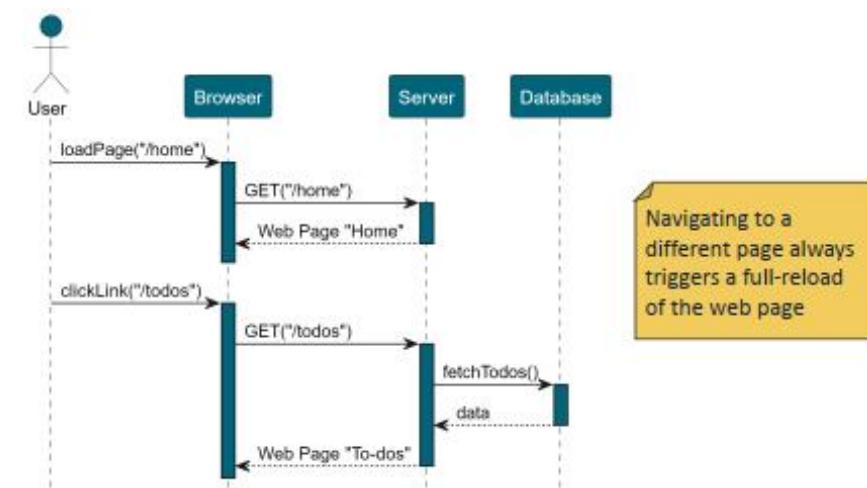
- analizza le **risorse statiche** da utilizzare nell'applicazione;
- compila i file Sass in **file CSS**;
- crea dei **bundle** di tutti i file CSS e di tutti i file JavaScript;
- effettua la **minificazione** dei bundle;
- effettua del **“fingerprinting”** sui bundle finali;
- sposta tutte le risorse nella cartella **/dist** (distribution: ciò che è pronto per essere mandato in produzione) ed aggiorna il file index.html con gli import giusti.

Applicazioni Web MULTI-PAGE

Lezione 18

L'approccio utilizzato fin ora si dice multi-pagina, in quanto vengono effettuate richieste al server, il quale crea una pagina HTML, e la manda indietro al Browser che la mostra all'utente. La ripetizione di questo processo per ogni pagina da visualizzare porta alla creazione di un'applicazione multi-pagina.

Più nello specifico, dunque, l'utente inserisce un URL nel suo Browser, che farà una richiesta HTTP GET al server, il quale manderà indietro una risposta contenente la pagina richiesta. Se l'utente clicca sui link interni alla pagina, allora il Browser effettua una nuova richiesta HTTP, e così via....

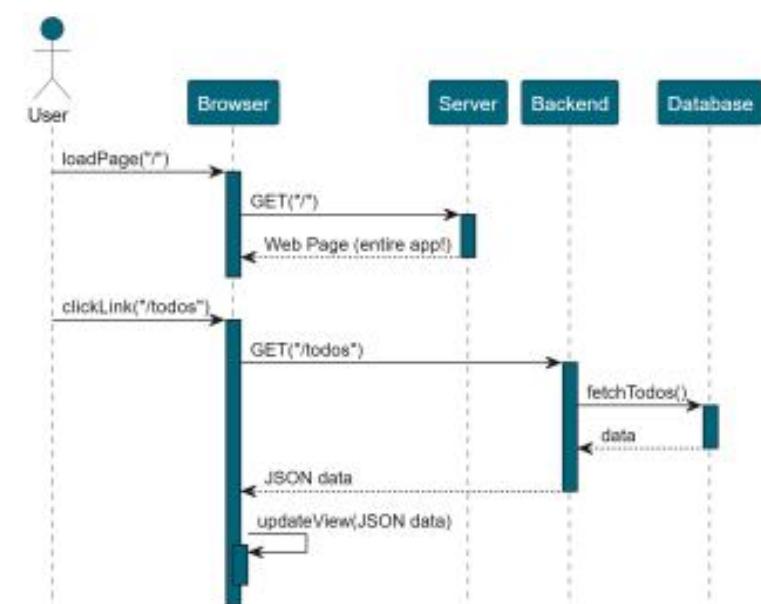


Applicazioni Web SINGLE-PAGE

Nelle applicazioni a pagina singola, se l'utente clicca su un link interno alla pagina, il Browser non avvierà l'intero ciclo di scaricamento e visualizzazione di una nuova pagina, bensì si limiterà a fare solo una richiesta di rete sul backend per ottenere gli oggetti da mostrare.

Il backend produrrà la sua risposta (come faceva anche nelle multi-page, magari tramite API REST), ed alla fine non manderà indietro un intero documento HTML da renderizzare, ma soltanto i dati da visualizzare (ad esempio, in formato JSON).

Il codice JavaScript, allora, deve aggiornare la pagina corrente per vedere le cose da fare. Si possono comunque utilizzare i link per continuare ad avere funzionalità specifiche.



ROUTING Lato Client

La navigazione tra le pagine (o meglio, le **viste** da mostrare) è gestita totalmente lato client.

Più nel dettaglio, il routing lato client si occupa di aggiornare l'URL nel Browser, di renderizzare la vista appropriata, e di interagire con le History API del Browser per consentire all'utente di navigare rapidamente avanti e indietro tra le varie viste.

Le **componenti principali** sono tre:

- **Router**: modulo che si occupa di mappare URL e viste;
- **View Rendering**: un meccanismo utilizzato per renderizzare le viste;
- **History Management**: un meccanismo per manipolare lo storico della navigazione e per aiutare l'utente a navigare con i bottoni "avanti" e "indietro".

Esempio di applicazione Single-Page

Il documento HTML dovrà essere uno solo e quasi vuoto: è lo script main.js che farà la gran parte del lavoro, in quanto lato server vengono mostrati solamente dei file statici.

Lo script **main.js** inizia poi a creare determinati elementi "principali" che andranno a formare lo scheletro della pagina, come ad esempio una barra di navigazione, un contenitore per il contenuto principale della pagina, ed un footer, assegnando a tutti questi un ID.

Gli oggetti **Route**, sono dotati di diverse proprietà, ognuna delle quali è un URL a cui viene associato a sua volta un ulteriore oggetto con due proprietà: il **titolo della vista**, ed una **callback che ne produce l'HTML** da mostrare all'interno del contenitore per quella specifica vista.

Il file js che si importa contiene il codice HTML (sotto forma di stringa) che rappresenterà l'intero contenuto della pagina. Si possono utilizzare pure dei template engine.

La funzione **router()**, per prima cosa, capisce se una delle varie viste definite nell'oggetto routes può essere applicata. A quel punto, si prova ad accedere alla proprietà di routes con tale pathname, che punta all'ultima parte dell'URL corrente. Se il path esiste, allora aggiorna il titolo della vista, e sostituisce il contenuto di content (l'oggetto creato all'inizio per la visualizzazione delle viste) col valore che ritorna la funzione render sulla vista specificata.

I **link** all'interno delle single-page application si dividono in due tipologie: quelli normali (che causano un nuovo caricamento di una pagina, tramite richieste HTTP), e quelli interni all'applicazione stessa (che effettuano soltanto un update della vista senza effettuare ulteriori richieste).

Tramite l'attributo **data-link** è possibile capire quali link devono o meno causare un nuovo caricamento della pagina.

Se l'elemento cliccato dall'utente possiede un attributo data-link, è possibile dire al Browser di inibire il comportamento predefinito, e di fare solamente delle cose specificate successivamente (come, ad esempio, interagire con le History API per consentire al Browser stesso di aggiornare l'ultima pagina visitata dopo il click sul bottone "indietro").

La funzione router() viene chiamata ogni volta che una pagina viene caricata del tutto (quindi a seguito dell'evento **DOMContentLoaded**) oppure al click sul tasto "indietro" (evento **popstate**).

COMPONENTI

Quando l'applicazione diventa complessa, gestirla in questo modo diventa molto difficile, poiché non è possibile **riutilizzare** le stesse funzionalità in viste diverse; per questo scopo, è possibile decomporre l'interfaccia utente in componenti, in maniera **modulare**, i quali saranno responsabili del **rendering della loro interfaccia in HTML** e della **gestione del loro stato interno**.

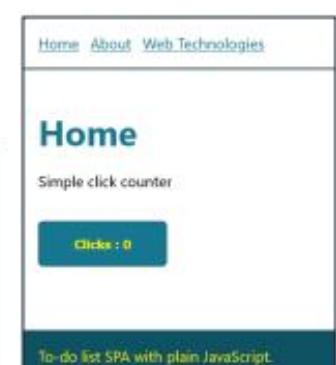
Le viste delegheranno tali azioni, dunque, ai singoli componenti, i quali possono anche essere utilizzati in viste diverse tra loro.

Si possono anche **creare** delle componenti partendo da zero, definendo una classe che estende HTMLElement ed avendo un attributo innerHTML. Lo stesso costruttore, dovrà registrare un handler per un certo evento.

Con il comando:

```
customElements.define("componentName", className);
```

stiamo quindi definendo un nuovo elemento HTML (del tutto nuovo rispetto a quelli del living-standard di default) che viene usato quando, all'interno di una pagina HTML dell'applicazione, viene trovato nel DOM.



```
import "../components/counter.js";
export default () => /*html*/
  <h1>Home</h1>
  <p>Simple click counter</p>
  <click-counter></click-counter>
`;
```

```
class Counter extends HTMLElement {
  constructor() {
    super();
    this.innerHTML = `<button>Clicks : ${count}</button>`;
    let btn = this.querySelector("button");
    btn.onclick = () => {
      btn.innerHTML = "Clicks : " + ++count;
    };
  }
}
```

Vantaggi di una Single-Page Application

Nonostante il backend venga complicato di molto, le SPA (Single-Page Applications) offrono diversi vantaggi rispetto alle applicazioni multi-pagina:

- **Esperienza utente più dinamica:** tutti i caricamenti, tranne il primo, sono istantanei;
- **Riduzione del lavoro del server:** le informazioni vengono recuperate solo quando necessarie, e viene ridotta anche la richiesta di banda;
- **Possibilità di funzionare offline:** una volta scaricata l'applicazione la prima volta, questa potrebbe funzionare senza rete, se non ci fosse il bisogno di recuperare ulteriori dati.

Sfide nell'implementazione di una SPA

Implementare una SPA da zero non è banale:

- la gestione dello stato è problematica;
- tenere aggiornata la UI non è facile;
- manipolare il DOM non è mai semplice e divertente;
- effettuare il routing lato client, per applicazioni grandi, potrebbe essere molto complicato;
- il rendering eseguito in questo modo non è molto efficiente, in quanto ogni volta sia butta via tutto per ricreare l'intero contenuto, quando idealmente si potrebbe aggiornare solo ciò che viene modificato.

Per ovviare a tutti questi problemi, ci vengono in aiuto i **Framework Frontend**, i quali forniscono dei modi standardizzati di costruire e gestire il lato frontend delle applicazioni.

Nota bene: per **eseguire** applicazioni single-page bisogna eseguire sia il backend che il frontend, separatamente.

Quality Assurance - Seminario by NTT Data

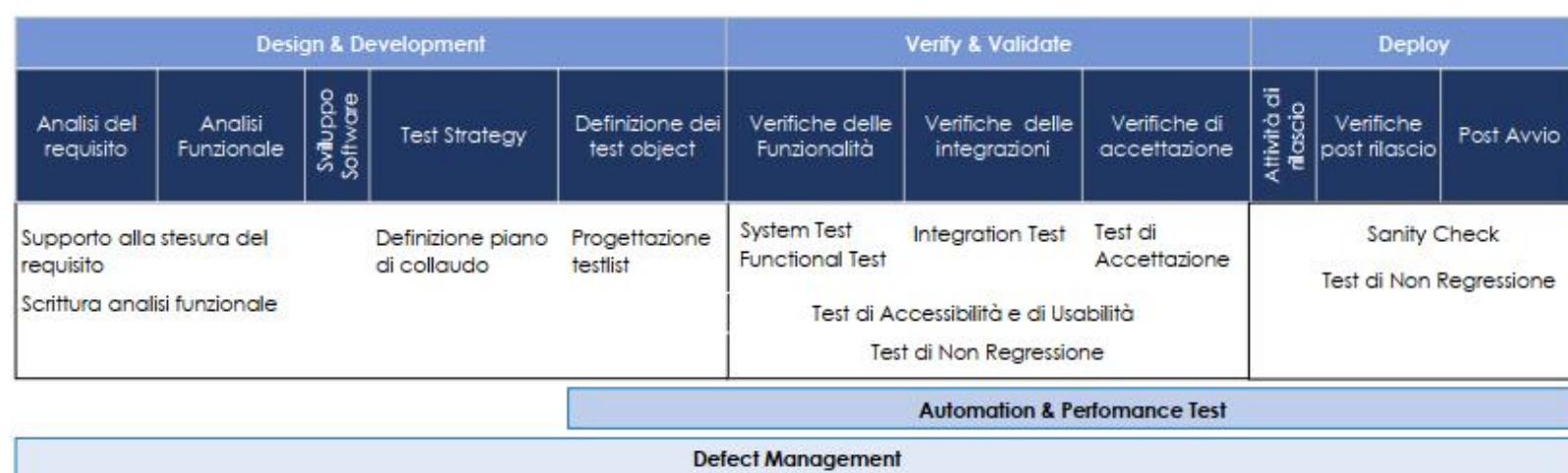
Lezione 19

(invertita con lez. 20)

Analisi del Testing: prendere delle casistiche, analizzarle, ed assicurarsi che il flow analizzato funzioni in maniera corretta:

- tirar fuori il requisito per bene;
- verifica e validazione del software, servendo ai clienti servizi di testing;
- garantire la qualità in fase di produzione.

Lo scopo è quello di convalidare dei bug o eventuali problemi, ed individuarne le vulnerabilità; vedere che il codice concordi con le specifiche; migliorare l'esperienza utente. Si formalizzano processi formali per la scoperta e la risoluzione dei problemi.



Ciclo di vita del Manual Testing: pianificazione, analisi, progettazione, implementazione, esecuzione UAT (User Accessibility Testing), reporting.

Differenza tra tipologia di Manual Testing: Waterfall ed Agile (molto più flessibile e divisione degli obiettivi in task).

I **test funzionali** rispondono alla domanda "COSA testare?", mentre i **test non funzionali** rispondono alla domanda "QUANTO BENE si comporta il sistema?".



Differenza tra **Happy Flow** (andato a buon fine) e **KO Flow** (in cui il risultato non viene raggiunto).

Metodologia dei test automatici: analisi dei requisiti, selezione degli strumenti, scrittura dei test script (sviluppo), esecuzione dei test, analisi dei risultati, manutenzione.

Il Testing Automation oggigiorno è di fondamentale importanza (ma non va sottovalutato neanche il contributo di quello manuale, che risolve spesso problemi i quali potrebbero non essere evidenti con l'insieme dei test automatici).



Si può utilizzare **XPath** (o CSS) per selezionare elementi o insiemi di elementi all'interno della pagina. XPath è un sovrainsieme di CSS (è più espressivo, non si è solo limitati a scorrere il DOM, ma consente anche di risalire all'indietro ad antenati, ecc...). Conviene utilizzare path RELATIVI, altrimenti ad ogni cambiamento, il path non funzionerebbe più.

Test Automation –XPath vs CSS Selector

 • //input[@id="pippo"]	 • input#pippo
• //input[@class="pippo"]	• input.pippo
• //input[@name="pippo"]	• input[name="pippo"]

Architettura TA (Testing Automation sul PROGETTO DI TEST, ossia un progetto apposito, diverso dall'applicazione, e fatto solamente con l'intento di testare):

- **test-case definition** (in maniera intuitiva e strutturata);
- **interaction** (design pattern del progetto: Page Object Model, organizzato in maniera settoriale e modulare);
- **drivers** (con i quali è possibile aprire ad esempio i Browser con impostazioni specifiche, apposite).

A parte, nell'architettura generale, ci sono anche i **Cloud Services**: ci danno la possibilità di utilizzare i loro dispositivi per effettuare testing più completo.



ANGULAR

Uno dei **framework frontend** (che ricordiamo essere insiemi predefiniti di software a supporto degli sviluppatori, e che forniscono una base sulla quale sviluppare in maniera semplificata delle applicazioni single-page) più usati, prodotto da Google.

Angular è fortemente orientato sul fornire strumenti già pronti per la produttività; è **fortemente opinionato**, ed include anche parti di routing e dependencies injection.

Creando un progetto Angular col comando **ng new angular-app --create-application** ci viene chiesto di scegliere il formato dei fogli di stile, ed automaticamente vengono generate moltissime componenti utili.

Col comando **ng serve** si abilita la modalità di sviluppo in live preview.

COMPONENTI

Sono dei mattoncini di base con i quali andiamo a comporre il frontend.

Ogni component è responsabile delle funzionalità e dell'aspetto **del suo relativo elemento**: codice, grafica, stile CSS, ecc...

Possiamo pensare ad una applicazione Angular come ad un albero di componenti, ognuna delle quali è una **classe JavaScript** annotata con la notazione **@Component**, tramite la quale è possibile specificare dei metadati per tale componente, passandoli sotto forma di oggetto.

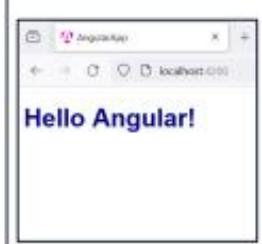
Per specificare ad Angular dove effettuare il rendering di uno specifico componente, si utilizzano dei **selettori**.

Stili e template possono anche essere specificati in file esterni, il cui path dovrà essere inserito nel component, tramite le regole **templateUrl** e **styleUrl**.

```
import {Component} from '@angular/core';

@Component({
  selector: 'app-root',
  standalone: true,
  template: "<h1>Hello Angular!</h1>",
  styles: "* {color: darkblue;}"
})
export class AppComponent {}
```

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AngularApp</title>
</head>
<body>
  <app-root></app-root>
</body>
</html>
```



Le componenti in Angular hanno un proprio **stato interno**, e le classi possono avere, quindi, delle **variabili di istanza**. Le espressioni contenute in **{{ }}** vengono valutate, convertite (se necessario) in stringhe, e renderizzate nel template.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  standalone: true,
  template: "<h1>Hello {{name.toUpperCase()}}!</h1>",
  styles: "h1 {font-family: sans-serif; color: darkblue;}"
})
export class AppComponent {
  name = 'Web Technologies';
}
```

Il caso più comune è che ci siano tanti componenti che verranno messi insieme in maniera gerarchica.

Con il comando **ng generate component nomeComponent** vengono creati diversi file in maniera automatica (**boilerplate code**: template HTML, classe di test, classe TypeScript, foglio di stile) tutti relativi e necessari alla component richiesta.

Il componente così creato può essere utilizzato e riutilizzato all'interno di altri componenti.

Per la gestione degli eventi nelle componenti (ossia, il bounding tra un evento ed il suo handler) si utilizza una sintassi a parentesi tonde, del tipo: **<button (eventType)="handlerFunction()">Text</button>**, in cui **handlerFunction()** potrebbe essere, ad esempio, un metodo interno del componente.

Il metadata **standalone** (che di default è impostato a true) garantisce che quel componente può esistere da solo, e che può importare direttamente al suo interno altri componenti e direttive per il suo template.

Per quanto riguarda il **controllo del flusso**, Angular supporta i classici template per la sintassi di **@if/@else** e **@for**.

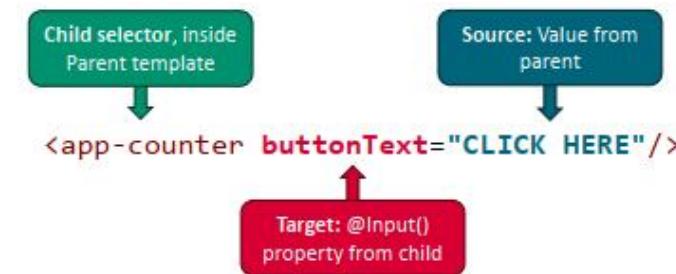
All'interno di un for utilizzato per il rendering di un template si può utilizzare il comando **track** per motivi di performance: serve a tenere traccia dell'associazione tra elementi e template, come una sorta di chiave primaria.

Comunicazione tra componenti

Quando una componente deve passare informazioni ad un suo componente figlio (o viceversa, quando il figlio deve comunicare col padre, ad esempio per il verificarsi di un evento), si può implementare un pattern di comunicazione utilizzando i decoratori **@Input()** ed **@Output()**.

@Input()

Può essere utilizzato per specificare che una determinata proprietà **potrebbe** essere passata da un genitore. Un antenato della componente con un elemento HTML taggato con @Input() può dunque passare informazioni al figlio impostando una proprietà sull'elemento HTML selezionato.

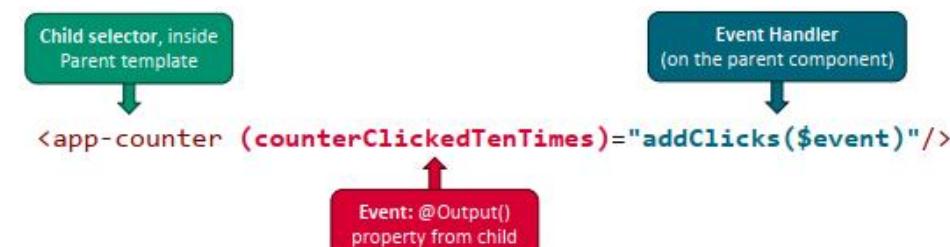


@Output()

Può essere utilizzato per creare un canale di comunicazione tra componente figlio e componente padre.

Sulla proprietà (taggata con @Output()) da passare bisogna assegnare un valore di tipo **EventEmitter**, per far capire che qualcuno più in alto nella gerarchia potrebbe averne bisogno, e vederla come un evento. Quando ciò deve accadere, si invoca il metodo **emit()** sulla proprietà.

Dunque, nel parent, per capire quando un evento si è verificato, innanzitutto c'è bisogno di avere il componente figlio, dopodiché si può utilizzare la solita sintassi per la gestione degli eventi (a patto che l'evento abbia lo stesso nome della proprietà taggata con @Output() del figlio).



Angular consente di definire delle **corrispondenze tra attributi** di elementi HTML dei template e **proprietà** del componente. Il nome dell'attributo viene specificato in parentesi quadrate [], ed il valore corrisponde al nome della proprietà da collegare.

Una cosa simile poteva essere fatta anche tramite **l'interpolazione** di template, con la differenza che lì l'espressione passata veniva trasformata in stringa, rendendo quindi impossibile il passaggio di proprietà di qualsiasi altro tipo.

```
<img [src]="imageUrl"/>
```

```
<h1>Hello {{name.toUpperCase()}}!</h1>
<app-counter [buttonText]="btnMsg"/>
<img [src]="imageUrl"/>
```

```
export class AppComponent {
  name = 'Web Technologies';
  btnMsg = "CLICK HERE";
  imageUrl = "https://picsum.photos/300/150"
}
```

Hello WEB TECHNOLOGIES!

CLICK HERE

The button was clicked 0 times.

* Button has never been clicked!



```

  <a routerLink="/" routerLinkActive="active" [routerLinkActiveOptions]="{exact: true}">
    Home
  </a> |
  <a routerLink="/counter" routerLinkActive="active">Counter</a>
</nav>
<router-outlet></router-outlet>
```

Lezione 21

FORMS

In Angular i form sono spesso il modo migliore per recuperare informazioni dagli input dell’utente, e possono essere gestiti in due modi:

Form template-driven: le associazioni tra componenti e dati avvengono in forma implicita, tramite direttive dei template.

La proprietà **[(ngModel)]** serve a dire ad Angular che il valore di quell’input deve essere sempre allineato ad una determinata proprietà;

la sintassi **[()]** (anche detta “banana in a box”) mostra che sostanzialmente si sta facendo da una parte il binding **([])** del valore di una proprietà del componente, che viene automaticamente aggiornata, e poi che stiamo comunque ascoltando il verificarsi di un evento **(())** a seguito del quale deve avvenire l’aggiornamento.

In questo modo, mentre l’utente inserisce l’input, i valori nel componente vengono aggiornati dinamicamente, e viceversa (**two-way binding**).

Form reactive: nell’approccio Reactive non abbiamo le proprietà all’interno di una component, ma si dichiara una sola proprietà (loginForm, ad esempio) che è un **form group**, ossia una cosa che si può importare direttamente da Angular.

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [FormsModule],
  template: './login.component.html',
  styleUrls: ['./login.component.scss'
})
export class LoginComponent {
  user: string = "";
  pass: string = "";
}

<h1>Login</h1>
<form>
  <label for="usr">Username:</label>
  <input id="usr" name="usr" [(ngModel)]="user"/>
  <label for="pwd">Password:</label>
  <input id="pwd" name="pwd" type="password" [(ngModel)]="pass"/>
</form>
<p>
  Username is <strong>{{user}}</strong>;
  Password is <strong>{{pass}}</strong>
</p>
```

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [ReactiveFormsModule],
  template: './login.component.html',
  styleUrls: ['./login.component.scss'
})
export class LoginComponent {
  loginForm = new FormGroup({
    //argument is initial value
    user: new FormControl(''),
    pass: new FormControl('')
  })
}

<h1>Login</h1>
<form [FormGroup]="loginForm">
  <label for="usr">Username:</label>
  <input id="usr" formControlName="user"/>
  <label for="pwd">Password:</label>
  <input id="pwd" formControlName="pass" type="password"/>
  <button type="submit">Submit</button>
</form>
<p>
  Username is
  <strong>{{loginForm.value.user}}</strong>;
  Password is
  <strong>{{loginForm.value.pass}}</strong>
</p>
```

```
<form [FormGroup]="loginForm" (ngSubmit)="onSubmit()">
  <input id="usr" formControlName="user"/>
  <input id="pwd" formControlName="pass" type="password"/>
  <button type="submit">Submit</button>
</form>
<p>
  Username is <strong>{{loginForm.value.user}}</strong>;
  Password is <strong>{{loginForm.value.pass}}</strong>
</p>
```

Indipendentemente dall’approccio scelto, la **submission** si gestisce sempre allo stesso modo, ossia come si gestiscono gli eventi, ma con la particolarità che in Angular l’evento non si chiama più submit, bensì **ngSubmit**.

SERVICES

Sappiamo che ogni classe dei nostri sistemi dovrebbe avere un’**unica responsabilità**: deve fare una sola cosa, e farla bene, per modularità. La conseguenza nello sviluppo con Angular è che ogni componente dovrebbe essere capace soltanto di garantire l’implementazione della sua funzionalità e lo stile per l’interfaccia utente.

Quando, tuttavia, abbiamo una funzionalità che può essere utilizzata in più parti del nostro programma, e che non è legata a qualcosa con cui l’utente interagisce in maniera diretta, non utilizziamo le componenti, ma un altro tipo di classe: i **Service**.

I Service sono soltanto **logica**, che può essere utile in molte classi del progetto, un po’ come i metadati per recuperare informazioni dalle REST API, per effettuare controlli, ecc...

Per creare dei servizi si utilizza il comando: **ng generate service serviceName**; questo comando produce un file **service.spec.ts**, utilizzato per testing, ed un file **.service.ts** che è il file in cui è effettivamente scritto il Service.

Dependency Injection

Angular utilizza il pattern Dependency Injection (DI) per permettere ai componenti di procurarsi dei service di cui hanno bisogno. L'obiettivo del DI è quello di separare due concern, ai fini di aumentare la modularità:

1. **istanziare** una certa dipendenza - ad opera del **dependency provider**, generalmente se ne occupa Angular;

2. **utilizzare** tale dipendenza - all'interno del **dependency consumer**, ossia il codice che scrive lo sviluppatore.

In generale, istanziare una dipendenza non è sempre banale, e non sempre ad un componente vogliamo delegare anche la responsabilità di istanziare la dipendenza. Dividere i task aiuta ad avere il codice più manutenibile e più facilmente testabile.

Le classi service che possono essere utilizzate altrove nell'applicazione, sono annotate con `@Injectable()`.

Iniettare classi in questo modo permette di utilizzare, per testing, delle **dipendenze fintizie** (dette "mock"), in quanto è possibile prevedere il risultato corretto (da inserire nel codice consumer) di cui si avrebbe bisogno dalla dipendenza (provider).

Un modo semplice per istanziare una dipendenza (anziché utilizzare il metodo `inject()` sul servizio) è quello di dichiarare, nel costruttore del componente, il servizio che si vuole importare.

All'interno del component viene automaticamente fatta l'injection della dipendenza.

```
import { Component } from '@angular/core';
import { CalculatorService } from '../calculator.service';

@Component({
  selector: 'app-home-page',
  standalone: true,
  imports: [],
  template: "<p>The sum is <strong>{{sum}}</strong></p>",
  styleUrls: ['./home-page.component.scss'
})
export class HomePageComponent {
  constructor(private calculatorService: CalculatorService){}
  sum = this.calculatorService.sum(1234, 4321);
}
```

Route Guards

Un task comune nello sviluppo di applicazioni è la **messa in sicurezza delle Route**, garantendo che solo alcuni utenti possano accedere a Route specifiche.

Angular supporta questo scenario con le **Route Guard**: delle **funzioni** che vengono invocate dal framework per capire se un utente è abilitato o meno ad attivare una certa route. Queste guardie si specificano come **proprietà** delle route stesse.

Per creare una guardia si usa il comando: `ng generate guard guardName`; vengono creati, anche in questo caso, un file per il testing ed un file TypeScript contenente il codice della guardia.

Quello delle Route Guard non è un vero sistema di sicurezza: il client sarà in possesso di tutta l'applicazione, quindi potrebbe aggirare le guardie, ma in un certo senso viene garantita la bontà dell'**esperienza utente**, in quanto senza modificare nulla tramite gli strumenti da sviluppatore, viene comunque bloccato l'accesso a determinate funzionalità dell'applicazione.

All'interno del file `app.routes.ts`, dove sono presenti tutte le route, si va ad inserire la guardia come proprietà della route specifica.

```
{ //in app.routes.ts
  path: "counter",
  title: "Counter",
  component: CounterPageComponent,
  canActivate: [authorizationGuard]
}
```

Una guardia può ritornare tre cose:

- **true**: quando la guardia viene soddisfatta e la navigazione può proseguire;
- **false**: quando la guardia non viene soddisfatta e la successiva azione non deve essere eseguita;
- un oggetto **UrlTree**: quando il Router deve effettuare un re-indirizzamento a quel determinato URL.

Nota bene: quando ci sono diverse guardie per una route, conviene sempre far tornare degli `UrlTree` per l'indirizzamento, perché usando `Router.navigate()` si rischia di avere delle race condition in caso di più guardie eseguite in parallelo.

HttpClient

Angular fornisce delle API per gestire le richieste di rete; queste API sono implementate nel service `HttpClient`, che va configurato utilizzando la dependency injection.

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class RestBackendService {
  constructor(private http: HttpClient) {}
  //This service can now make HTTP requests via 'this.http'.
}
```

Angular fa spesso uso degli **Observable**, ossia degli oggetti che a differenza delle promesse (le quali possono essere risolte o rifiutate una sola volta), emettono in maniera asincrona, potenzialmente, **più valori nel tempo**.

Questi oggetti observable vengono dunque ritornati dai metodi presenti in HttpClient per gestire le varie tipologie di richieste HTTP.

Lavorando con gli observable, si utilizza il metodo **subscribe()** per fornire delle **callback** da eseguire a seguito di un evento. I metodi delle callback sono tre:

- **next(val)**: eseguito quando l'observable emette un valore;
- **error(err)**: eseguito quando l'observable emette un errore (e vale anche come completamento dell'observable);
- **complete()**: eseguito quando l'observable ha finito di compiere il suo lavoro e non deve emettere ulteriori valori.

```
import { Observable, Subscriber } from "rxjs";

const observable = new Observable<number>( subscriber => {
  helper(subscriber, 1);
});

function helper(subscriber: Subscriber<number>, level: number){
  setTimeout( () => {
    let num = Math.random();
    if(level > 3){ subscriber.complete(); return; }
    if(num < 0.8) {
      subscriber.next(num);
      helper(subscriber, level+1); //recursion
    } else { subscriber.error(new Error(`You were unlucky! ${num}`)); }
  }, 1000);
}

observable.subscribe({
  next(value){ console.log(`Observer emitted value ${value}`) },
  error(err) { console.log(`Observer emitted an Error: ${err}`) },
  complete() { console.log("Observer finished emitting values") }
});
```

@luigi → D/O/T/W/2/e/2/a/src \$ node .\observables_example.js
 Observer emitted value 0.09009756551320991
 Observer emitted an Error: Error: You were unlucky! 0.8901570954122466
 @luigi → D/O/T/W/2/e/2/a/src \$

Interceptors

Quando si configurano i provider della Dependency Injection di HttpClient, si possono customizzare molte cose, come ad esempio la scelta dell'usare le Fetch API (**withFetch()**) al posto di XMLHttpRequest, oppure la configurazione degli Interceptor.

Gli Interceptor sono una sorta di **middleware** per HttpClient, e si applicano alla richiesta d'uscita.

Non sono altro che **funzioni** che prendono in input la richiesta corrente in uscita, e **next()**, che è il prossimo step della pipeline di interceptor. Possono essere utili se si vuole sistematicamente aggiungere a tutte le richieste d'uscita un header, come ad esempio un token.

```
function log(req: HttpRequest, next: HttpHandlerFn): Observable {
  console.log(req.url);
  return next(req);
}
```

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(
      withInterceptors([loggingInterceptor, authInterceptor])
    )
  ]
}
```

Change Detection

In Angular esiste un meccanismo per effettuare modifiche nel momento in cui avviene un cambio nello **stato dell'applicazione**, che è la **routine** di change detection: questa routine naviga per tutti i componenti a partire dell'AppComponent, e poi per ognuno di questi controlla se il suo stato interno sia cambiato. Se questo è cambiato ed è utilizzato in qualche template, va a cambiare la parte di template che è stata modificata.



Signals

Si possono immaginare come delle normali proprietà delle componenti, che però hanno il potere di **notificare** tutti i consumatori nel momento in cui il loro valore cambi, consentendo una sorta di **programmazione reattiva**. Usare i signals permette ad Angular di ottimizzare ancora di più il meccanismo di change detection.

I segnali possono essere **writable** (sovrascrivibili) oppure **read-only** (di sola lettura). Questi vengono creati con il metodo **signal(val)**, a cui si passa un valore iniziale, dopodiché (quelli writable) possono essere settati o aggiornati attraverso i metodi **.set()** e **.update()**.

```
export class CounterSignalComponent {
  count = signal(0); //initial value is zero
  handleClick() { this.count.update(value => value + 1); }
  handleReset() { this.count.set(0); }
}
```

Particolari signal di sola lettura vengono detti **Computed Signals**: questi derivano il loro valore da quello di altri signal. Per crearli si usa il metodo **computed()**. Il framework, automaticamente, si prenderà carico di mantenere tutti i valori aggiornati in maniera reattiva.

```
export class CounterSignalComponent {
  count = signal(0); //initial value is zero
  double = computed(() => this.count() * 2);
  //rest of the component omitted
}
```

Un **Effect** è una funzione che viene eseguita ogni volta che un signal cambia valore.

La funzione passata viene eseguita sempre una prima volta, dopodiché ogni volta che uno dei signal al suo interno cambia, viene eseguita nuovamente.

```
export class CounterSignalComponent {
  count = signal(0); //initial value is zero
  double = computed(() => this.count() * 2);

  constructor(){
    effect(() => {
      console.log(`Count value changed: ${this.count()}`);
    })
  }

  handleClick() { this.count.update(value => value + 1); }

  handleReset() { this.count.set(0); }
}
```

Console log output
Count value changed: 0
Count value changed: 1
Count value changed: 2
...

Tailwind CSS

Lezione 22

All'interno del progetto, potrebbe essere utile l'utilizzo di Tailwind CSS: un framework CSS molto utile per lo sviluppo di pagine web, in quanto comprende **classi** come "flex", "text-center" e simili che possono essere unite per creare un design della pagina in maniera semplificata, già **all'interno del markup**.

Tailwind CSS può essere importato nel progetto aggiungendo "**tailwindcss**" alle devDependencies ed aggiungendo un file di configurazione.

The screenshot shows a code editor interface with two main sections. On the left, under 'OPEN EDITORS', there is a list of files: tailwind.config.js (highlighted), angular todo list, .editorconfig, .gitignore, angular.json, package-lock.json, package.json, README.md, tailwind.config.js, tsconfig.app.json, tsconfig.json, tsconfig.spec.json, and angular todo list rest. Under 'TO-DO LIST SPA WITH ANGULAR EXAMPLE', there is a folder named 'angular todo list' containing 'src', '.editorconfig', '.gitignore', 'angular.json', 'package-lock.json', 'package.json', 'README.md', 'tailwind.config.js', 'tsconfig.app.json', 'tsconfig.json', 'tsconfig.spec.json', and 'angular todo list rest'. On the right, the content of 'tailwind.config.js' is displayed:

```
angular todo list > JS tailwind.config.js > ...
1  /** @type {import('tailwindcss').Config} */
2  module.exports = {
3    darkMode: 'class',
4    content: [
5      './src/**/*.{html,ts}',
6      './src/index.{html,ts}'
7    ],
8    theme: {
9      extend: {},
10    },
11    plugins: [],
12  }
13
14 |
```

SICUREZZA in una Web Application

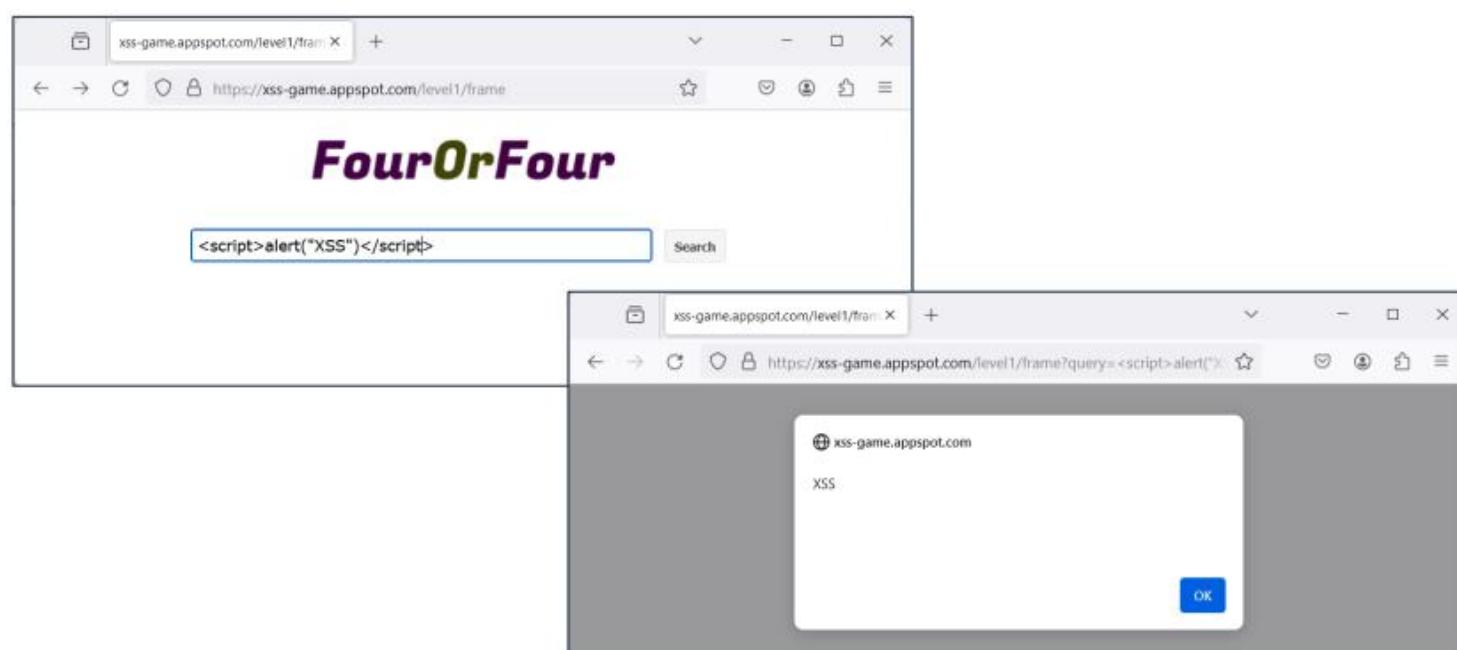
L'obiettivo di questo ramo è riuscire a garantire **confidenzialità** ed **integrità** dei dati, nonché assicurarsi che questi siano resi **disponibili** (solo) a chi di dovere.

Si può lavorare sulla sicurezza di un'applicazione web sia a livello di rete che a livello di applicazione:

- Sicurezza a **livello di Rete**: ci si concentra sulla crittografia dei dati (HTTPS), sui filtri dei firewall, sui log di pacchetti che arrivano sulla rete, ecc...; l'obiettivo è dunque quello di proteggere i canali di comunicazione e le infrastrutture di rete;
- Sicurezza a **livello di Applicazione**: si studiano quali sono le misure di sicurezza che si possono garantire da sviluppatori web: autenticazione ed autorizzazione, validazione di input, gestione corretta delle sessioni, confidenzialità dei dati...

Cross-Site Scripting (XSS)

Questa vulnerabilità consiste nella possibilità di **inserire codice client-side** malevolo all'interno di pagine web. Ciò accade quando, ad esempio, non viene svolta la validazione dell'input (magari senza effettuare l'escaping delle HTML Entities).



Codice client-side iniettato in questo modo viene eseguito dal server sotto forma di codice “**trusted**”: potrebbe causare qualsiasi tipo di danno, come ad esempio la manipolazione dei cookies, l'accesso al localStorage ed al sessionStorage, l'esecuzione di determinate azioni involontarie (come il redirect ad un URL malevolo)...

Per mitigare vulnerabilità di questo tipo, si deve lavorare su ogni cosa che possa minimamente essere **toccata dall'utente**, inserendo tutte le operazioni di escape ed i necessari controlli (sanitize della pagina web).

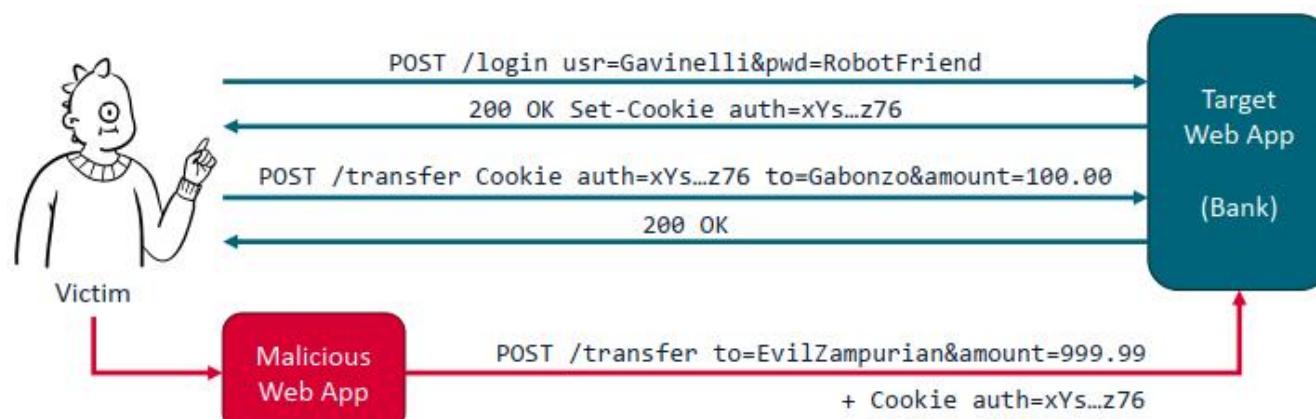
Il modo migliore non è quello di avere un approccio basato sul filtraggio e basta, ma conviene **sanitizzare gli input** nel momento in cui questi vengono inseriti all'interno della pagina. Ci sono soluzioni ad hoc in tutti i principali framework, come ad esempio la libreria **sanitize** di Node.js.

Anche **Angular** ha un suo security model per mitigare gli attacchi XSS: per lui, qualsiasi dato che viene scambiato all'interno di un **template**, è trattato come **non affidabile** di default. In questo caso, Angular ne fa automaticamente l'escaping, sanitizzandolo. Se vogliamo che su determinati dati non venga effettuato alcun controllo, si può utilizzare il metodo **bypassSecurityTrustHtml(value: type): SafeHtml** di DomSanitizer.

Cross-Site Request Forgery (XSRF o CSRF)

L'obiettivo di questi attacchi è quello di far eseguire delle **azioni involontarie** agli utenti sfruttando il fatto che questi siano **autenticati** su applicazioni web non sicure.

Queste azioni potrebbero riguardare cambiamenti di stato delle operazioni in corso, cambio delle credenziali associate ad un account, ecc...



I passaggi di questo attacco potrebbero essere i seguenti:

1. L'utente clicca su un **link malevolo** dal browser all'interno del quale è autenticato in quel momento;
2. La pagina malevola invia un **form** alla web app "legale", proprio come se fosse stato l'utente;
3. Il browser della vittima si accorge che la richiesta è diretta verso la pagina "legale", quindi aggiunge alla richiesta i **cookie di autenticazione**, che rendono l'utente capace di proseguire con azioni involontarie all'interno della pagina web.

Per mitigare gli attacchi CSRF si utilizza un approccio detto **Synchronizer Token Pattern**: l'idea è che l'app web, nel momento in cui crea una sessione per l'utente, generi anche un **token**, salvandolo all'interno della sessione stessa. Quando c'è un form da proteggere da attacchi di questo tipo, l'app aggiunge il token come primo **input segreto** (hidden) di questo form; nel momento in cui viene effettuata la submit, l'applicazione può controllare se il token ricevuto dal form corrisponde o meno a quello salvato all'interno della sessione.

Molti framework includono **capabilities** per l'implementazione delle contromisure agli attacchi CSFR, e tra questi anche Angular e Spring, il quale lo fa di default tramite un meccanismo basato su token.

SQL Injections

Consiste nell'**iniettare codice SQL** all'interno di statement o query fatte **su una base di dati**. Il codice iniettato nasce da input utente non progettati per bene.

Attacchi del genere possono risultare in leak di dati sensibili, esecuzione di codice sul database, ecc...

Una delle vulnerabilità più grandi è che, se ad esempio usassi i modelli di Sequelize per effettuare una query su un database, i parametri sarebbero **inseriti direttamente all'intero della query** che deve essere eseguita; ciò comporta che se inserissi un input contenente della sintassi SQL, questa potrebbe provocare azioni malevoli sulla base di dati.

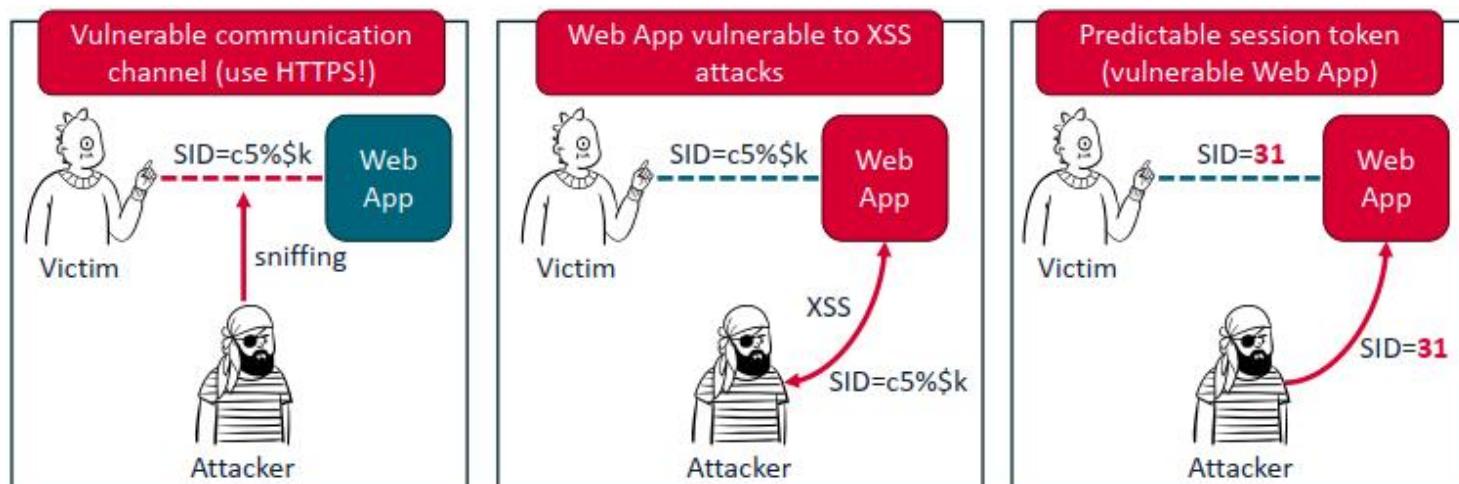
Un esempio potrebbe essere la possibilità di loggarmi con l'account di un altro utente senza conoscere la sua password:



Per evitare vulnerabilità di questo tipo, si possono utilizzare statement con **query parametriche**, oppure **stored procedures** su database.

Session Hijacking

Attacchi che mirano a **compromettere il token di sessione**: o si sottrae all'utente il suo token facendo (ad esempio) dello **sniffing**, oppure predilige (nei casi in cui una Web App vulnerabile non utilizza funzioni di hashing o randomiche). Se l'utente non utilizza HTTPS o altri **protocolli sicuri**, gli altri utenti possono vedere tutto il traffico in chiaro.



Validare l'input lato client è una buona pratica che migliora l'accessibilità, ma è fondamentale farlo anche lato server, in quanto lato client si può aggirare tutto mandando delle richieste HTTP fatte a mano.

Servizi di sicurezza forniti dai Web Browser

I Browser moderni hanno l'enforcement di un **modello di sicurezza** abbastanza **stringente**: tutto è eseguito all'interno di una sandbox, ad esempio, oppure ci sono delle policy che il Browser segue di default.

Same-Origin Policy

Di default, nei Browser moderni, documenti e/o script possono interagire soltanto con risorse che provengono dalla stessa origine: l'**URL** deve avere lo **stesso protocollo**, la **stessa porta**, e lo **stesso hostname**.

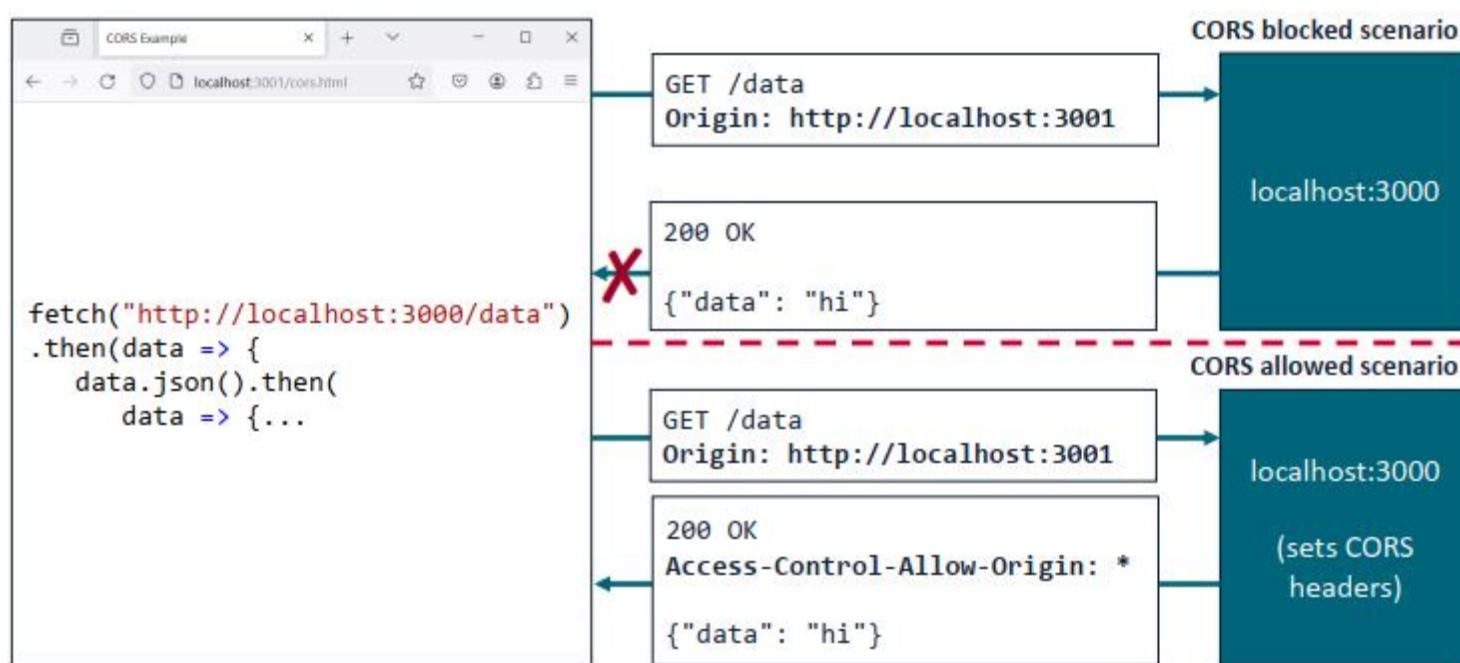
Anche se tra Browser diversi esiste qualche minima differenza, la Same-Origin Policy si applica a tutti se per il retrieve dei dati da codice JavaScript si utilizzano funzioni quali `fetch()` ed `XMLHttpRequest()`.

Quando si tenta di accedere ad una risorsa che non ha la stessa origine, il Browser manda comunque la richiesta e riceve la risposta in maniera corretta, ma è lui stesso che decide di non offrire accesso al contenuto della risposta ricevuta.

Cross-Origin Resource Sharing (CORS)

A volte è necessario accedere a contenuti provenienti dall'esterno di una Web App; il meccanismo che permette di controllare quando è possibile **accedere a risorse esterne** si chiama CORS: questo permette ad un server di dichiarare se alle sue risorse possono accedere tutti o soltanto persone provenienti da una certa origine (tramite un header CORS nelle risposte, che deve matchare con l'header di Origine inserito dal client che ha effettuato la richiesta).

Dunque, il blocco nella comunicazione non avviene in uscita, bensì in entrata: bisogna vedere se nella risposta è presente l'header necessario. Si tratta di un meccanismo per proteggere pagine web fideate da attacchi come, ad esempio, gli XSS.



SOFTWARE VERIFICATION

Lezione 24

Si tratta di tutte quelle pratiche che hanno lo scopo di assicurarsi che il sistema prodotto **soddisfi le specifiche** richieste. Si divide in **verifiche dinamiche** (testing del software) e **verifiche statiche** (leggibilità e bontà del codice, analisi statiche, verifiche automatiche...). L'obiettivo è quello di **trovare** quanti più **bug** possibili prima che il prodotto venga rilasciato.

SOFTWARE TESTING

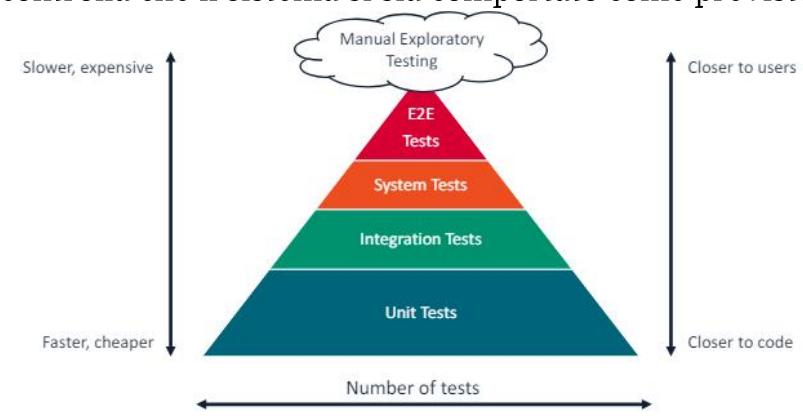
Un software test è una **sequenza di azioni** progettata per valutare una particolare funzionalità del nostro software, o un suo aspetto.

Prevede una fase di **preparazione** (in cui si controlla che le condizioni iniziali siano tutte soddisfatte), una fase di **azione** (in cui vengono eseguite una o più azioni), ed una fase di **asserzioni** (dove si controlla che il sistema si sia comportato come previsto).

Il testing può essere svolto a diversi livelli:

- **test di unità**: ad esempio, si testa una classe oppure un singolo metodo;
- **test di integrazione**: per vedere se le unità collaborano bene tra di loro;
- **test di sistema**: aggregato di tutti i moduli;
- **testing End-to-End (E2E)**: dal punto di vista dell'utente finale.

Salendo nella piramide, i test diventano via via più lenti.



Web Application Testing

Ci sono diverse particolarità nel testing di applicazioni web: il test di unità dovrà fare i conti con codice asincrono, oppure quando dobbiamo testare un component dobbiamo avere a che fare con i framework... i nostri componenti poi dipendono da services, quindi anche dall'esterno.

Altre sfide dei test End-to-End sono, inoltre, la fragilità e la poca affidabilità.

Angular supporta i test di unità con un framework detto **Jasmine**. Nel momento in cui viene creato un service, viene anche generato un file **.spec**, che è un file di test.

Un **file di test** creato con Jasmin è fatto in un modo particolare: viene creato innanzitutto utilizzando un metodo **describe()**, che serve a generare una suite di test, ed a cui si passa un nome ed una callback.

All'interno della callback si vanno a definire i test.

Viene creata la variabile **service**; con **BeforeEach()** ci si assicura che quel codice venga effettuato prima di tutto il resto.

TestBed è il modo che Angular fornisce per interagire con i propri componenti in fase di testing: è una raccolta di utility.

Il metodo **it()** definisce un singolo spec.

Poi ci si assicura con le **asserzioni** che le cose vadano effettivamente come ci si aspetta, e quindi ad esempio che il service sia stato creato, e che svolga correttamente il suo lavoro.

Da riga di comando si può utilizzare il comando **ng test** che effettua la build dell'app ed avvia un programma interattivo che si chiama **Karma**, il quale esegue tutti i file **.spec.ts**.

```
import { TestBed } from '@angular/core/testing';
import { CalculatorService } from './calculator.service';

describe('CalculatorService', () => {
  let service: CalculatorService;

  beforeEach(() => {
    TestBed.configureTestingModule({}); // Create a module with an empty
    // configuration. This is equivalent to doing nothing.
    service = TestBed.inject(CalculatorService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('should sum two positive numbers correctly', () => {
    expect(service.sum(42, 58)).toEqual(100);
  });
});
```

describe() creates a set (a.k.a. **suite**) of specs (or tests). Same idea as a JUnit Test Class. It groups related tests.

TestBed is the most important Angular Testing Utility. Provides methods to create components and services in unit tests.

it() defines a single spec. Same idea as a **@Test** method in JUnit.

Test Doubles

Spesso, le cose da testare dipendono da altri **servizi** o da **componenti esterni**, come ad esempio HttpClient, RestBackend Service, ed altri. Fare testing in questi scenari diventa un po' più complicato: se succede qualcosa che non mi aspetto, non ho la certezza di sapere da dove provenga il bug.

Al di là del problema della **localizzazione del bug**, c'è anche il fatto che in molti casi potremmo non aver modo di utilizzare delle dipendenze reali all'interno dei nostri service (come ad esempio, se ci sono valori che cambiano in base alla temperatura corrente della stanza in cui si effettua il test).

Per risolvere, si dovrebbe effettuare del **testing in isolamento**: ciò che testiamo non dovrebbe dipendere da unità esterne.

La soluzione è quella di rimpiazzare le dipendenze reali con un **Test Double**: un oggetto che non è quello reale, ma che si comporta in maniera simile, e che è controllabile dallo sviluppatore.

Il meccanismo di Dependency Injection è fondamentale quando c'è da utilizzare dei Test Double: ci consente di cambiarli senza dover modificare minimamente il nostro codice.

Fortunatamente, i test di framework offrono supporto anche per la creazione automatica dei Double.

In Java, ad esempio, esiste Mockito. Jasmine utilizza invece **Spies**: una Spy può effettuare il mock di qualsiasi funzione.

Invece di utilizzare un vero RestBackendService, si utilizza un qualcosa che ritorna un oggetto statico.

```
let todos = [{id: 1, todo: "foo"}, {id: 2, todo: "bar"}];
restBackendSpy = jasmine.createSpyObj('RestBackendService', ['getTodos']);
restBackendSpy.getTodos.and.returnValue(of(todos));
```

Si può effettuare test sul frontend in questo modo anche quando il backend ancora non esiste.

È anche possibile ottenere degli indicatori di quanto è stata testata bene l'applicazione, osservando le **code coverage metrics** che vengono generate nella cartella `/coverage`.

Testing END-TO-END (E2E)

Ogni test in un contesto End-to-End mira ad investigare i **flussi di utilizzo del sistema** dal punto di vista dell'utente finale.

Anche nel caso in cui l'applicazione fosse un'app REST, in cui quindi gli utenti sarebbero dei programmi esterni che inviano richieste HTTP, un test E2E dovrebbe verificare che le risposte siano effettivamente corrette in base alle specifiche.

In caso di app web, invece, gli utenti finali sono delle persone che utilizzano un Browser: un test E2E in questo contesto interagisce con la pagina web e si assicura che tutti i cambiamenti avvengano come previsto a seguito delle interazioni.

Per ogni **scenario**, nei test E2E (sia automatizzati che manuali), vanno eseguiti degli **step**, uno alla volta. Ogni volta che deve essere eseguito uno step dello scenario, ci sono due task importanti:

1. capire **con quale elemento** bisogna interagire;
2. effettuare **l'interazione**.

Testing E2E MANUALE ed AUTOMATIZZATO

Il testing E2E può essere svolto in maniera manuale (da parte di tester umani a cui viene dato una lista di step da seguire per attuare gli scenari) oppure in maniera automatizzata (i tester sviluppano delle suite di test, software, che simulano automaticamente gli scenari, e che siano ripetibili).

Il **test manuale** ha come vantaggio il fatto che non ci sia bisogno di un informatico o di un programmatore per eseguire i test, però sono molti dispendiosi in ambito di tempo, ed è molto probabile che vengano effettuati errori. Inoltre, non seguono minimamente un concetto di scalabilità, in quanto dovrebbero essere ripetuti per ogni singola modifica dell'applicazione.

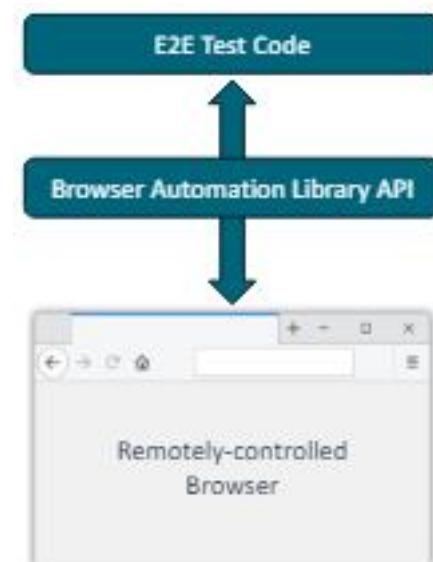
Il **test automatizzato** ha dei costi iniziali alti, ma può essere ripetuto quante volte si vuole (anche se potrebbe necessitare spesso di manutenzione in caso l'app venga modificata). Test di questo tipo possono essere fragili ed inconsistenti.

Per effettuare test automatizzati vengono utilizzate delle **librerie** che consentono di **navigare verso precisi URL o verso elementi HTML specifici**.

Queste librerie sono utili anche per eventuali task di Scaping e Crawling.

Per trovare l'elemento con cui interagire, ci sono diversi approcci:

- **Coordinate absolute**: si utilizzano le coordinate esatte in cui cliccare all'interno della pagina;
- **Localizzatori visual-based**: si utilizza un approccio visuale che consente di identificare gli elementi in base al loro aspetto, magari con degli screenshot (un tool molto comodo è SikuliX);
- **Localizzatori basati su layout**: si identificano elementi in base alle proprietà del layout, come ad esempio selettori CSS o XPath.



Due sfide per chi effettua testing E2E di app web sono la fragilità e l'inconsistenza:

Fragilità

Ognuno dei metodi per trovare i localizzatori ha vantaggi e svantaggi.

La fragilità è quel problema che, per via di cambiamenti anche minimi nell'applicazione, può rompere dei localizzatori che precedentemente funzionavano, facendo fallire il test, e rendendo necessarie operazioni di manutenzione.

Degli esempi possono essere la **modifica della pagina** quando si utilizzano coordinate assolute, oppure un **cambiamento dei colori** quando si utilizzano localizzatori visuali, o ancora l'**aggiunta di un altro bottone** al posto di quello precedente quando si utilizza un selettore CSS.



Per creare dei locator robusti e che non vadano modificati ad ogni cambiamento dell'applicazione, si dovrebbe tentare di farli né troppo generici, né troppo specifici, altrimenti basterebbe un minimo cambiamento per rompere il selettore.

Ogni tipologia di selettore ha le sue fragilità e le sue robustezze. In generale, quelli basati su layout sono i più robusti, e per questo, i più utilizzati.

Inconsistenza

Non è raro che lo stesso test passi alla prima esecuzione senza problemi, per poi fallire alla seconda esecuzione.

Questi problemi sono causati spesso dal **non-determinismo** del caricamento degli elementi o del recupero delle informazioni.

Per questo motivo, anche questi test dovrebbero essere eseguiti in **isolamento**.

Per ottenere isolation test in E2E si può utilizzare o un **approccio "from scratch"**, in cui si parte da zero, oppure c'è il bisogno di ricordarsi di effettuare **operazioni di clean-up** tra un test ed il successivo.

Tools per testing E2E



Playwright

Playwright è un framework open-source per l'automatizzazione dei Browser, sviluppato da Microsoft nel 2020.

È cross-platform e cross-browser, ed utilizza un approccio asincrono, orientato agli eventi.

È scritto in JavaScript, ed ha API utilizzabili su tutti i principali linguaggi di programmazione.

Si installa con il comando: **npm init playwright@latest**.

Playwright consente agli sviluppatori di interagire in maniera remota con i Browser, localizzando elementi interni alle pagine, interagendo con questi, e generando dei test in maniera molto rapida tramite la funzione **test()**, che prende in input un titolo e la funzione asincrona da eseguire.

```
import { test, expect } from '@playwright/test';

test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Expect the page has a title matching a substring.
  await expect(page).toHaveTitle(/Playwright/);
});
```

Cattura e Replay

Esistono degli approcci che ci consentono di generare codice E2E eseguibile in maniera automatica e senza doverli scrivere da zero.

L'idea è che invece di scrivere i test a mano, si utilizzino dei tool che **registrano le nostre interazioni con l'applicazione web**, e che **generano dei test automatici**.

Test fatti in questo modo potrebbero essere più facili e meno affidabili di quelli scritti a mano, ma sono comunque un buon punto da cui iniziare.

Uno di questi tool è un'estensione di VS Code, chiamata **VS Code Playwright extension**.

Durante questo corso, nell'a.a. 2023/2024, non è stata tenuta dal Docente la lezione su **CMS** e **GraphQL**.

Lezione saltata

The end of Web Technologies

Disclaimer

La maggior parte delle immagini presenti in questo file sono state prese dalle slides ufficiali del corso, create dal Professor Luigi Libero Lucio Starace.

Questi appunti non intendono in alcun modo sostituire il corso da tenere in presenza, in quanto non sarebbero sufficienti al superamento dell'esame data la mancanza di esempi e di ulteriori dettagli, ma hanno lo scopo di essere d'aiuto agli studenti nei giorni di ripetizione precedenti l'appello.

Grazie per l'attenzione.

Fabrizio Formicola