# HTTP

HTTP è un protocollo State-Less.

**State-less**: La sessione logica *http* non coincide con la sessione reale *TCP*, che invece è composta da tante sessioni. Perciò *http* non si ricorda chi siamo, cioè non conserva informazioni su chi siamo.

HTTP è un protocollo client-server. La comunicazione avviene attraverso una richiesta del client, al quale segue una risposta del server.

###### *URL (Uniform Resource Locator):*

La richiesta di una risorsa avviene tramite l'URL, ed è così composta:

**[PROTOCOL]://[DOMAIN]:[PORT]/[RESOURCE]?[QUERY]**

Esempio di *URL:* <http://pics.photos/200/300/?random>

La porta è ormai standardizzata, quindi omessa. 'random' è la query (che ad intuito corrisponde a una get).

###### *Metodi (Verbi) dell' HTTP:*

L'*http* permette la comunicazione attraverso dei verbi (o metodi):

* **Get**: richiede la risorsa.
* **Post**: viene utilizzata per delle richieste di creazione risorse.
* **Put**: update una risorsa esistente
* **Delete**: elimina una risorsa esistente.

*Put e delete* sono considerate specializzazioni di *Post* e vengono usate dagli sviluppatori. *Get e post* vengono usati invece dai client.

Il verbo *post*, a differenza della *get*, anziché mettere tutti i dati nel body, come fa la *get*, (ad esempio se vogliamo fare l'upload di una immagine) definisce una fase di preload in cui inserire i dati, con renderizzazione del body.

Una ulteriore differenza tra *get* e *post* è che se vogliamo che l'utente possa conservare l'indirizzo della pagina su cui sta lavorando, allora impostiamo l'accesso alla pagina come *get*, se invece vogliamo che non possa conservarla allora impostiamo l'accesso come *post*.

##### Tramite url e verbi si può quindi cominciare la comunicazione con il server...

Ma la risposta del server è composta dall'url e da un codice di stato che comunicano lo stato della richiesta:

###### *Codici di Stato:*

Si dividono in 5 classi...

1XX – info message

2XX – messaggi di successo

3XX – cache messages

4XX - (tipo 404 bad request) client error mesages.

5XX - (i peggiori) server error messages.

**Front-end**: è il lato client, è una parte di codice che permette di definire una applicazione grafica che permette di interagire tramite il protocollo *http* con altri servizi back-end tipo server e database.

E' composta dalle tecnologie html, css e js. Che definiscono la struttura, come l'interazione grafica deve essere definita con l'utente, e un linguaggio di programmazione che riesca a gestire la nostra interfaccia e l'interazione con l'utente.

**Back-end**: è una macchina che sta disponendo un servizio, un *server*, tramite protocollo http.

# HTML

Non è un linguaggio di programmazione, è un linguaggio di markup. Non dispone di alcuni costrutti della classica programmazione logica (if e while), per questo non permette una programmazione logica, ma definisce per lo più la struttura della pagina web.

### Browser web:

Un web browser richiede e interpreta richieste *http*.

### Progetto di un applicativo web:

Come organizzare il progetto

* Una cartella che contiene il progetto e file html.
  + Un cartella *assets* che contiene immagini, css e js.
    - Una cartella di css
    - Una cartella js
    - Una cartella img
  + Index.html

L'index è l'entry point (una sorta di main), il punto di inizio.

**Il <!DOCTYPE>** è una dichiarazione che serve ad aiutare il browser a capire cosa c'è sotto e come lo deve interpretare.

Il doctype è la prima riga che vediamo all'interno di un file html.

###### *HTML page structure:*

All'interno dell'html troviamo due figli: head e body. All'interno del head inseriamo dati META e un figlio; title, che definisce il titolo della pagina web.

All'interno del body troviamo gli ELEMENTS.

* **Head**: contiene tutte le meta informazioni della pagina. Tipo il titolo, la lingua, se è per mobile, se può essere zoomata.
* **Body**: inserisco tutti gli elementi che voglio renderizzare, scritte, bottoni ecc...

Un elemento html è defito da **tag <tag>.**

Gli elementi sono divisi in due categorie, elementi InLine e a Blocco.

* **InLine**: prende invece la larghezza richiesta dal contenuto.
* **Blocco**: di default prende sempre il 100% di larghezza della nostra pagina.

**L'HTML è case Insensitive e gestisce i suoi elementi con una struttura ad albero.**

### Elementi testuali dell'HTML:

Sono tutti con chiusura.

* Testo semplice
* <p>*Paragrafo*</p>
* <span>*Span*</span>
* <b>*Bold*</b>
* <i>*Corsivo*</i>
* <small>*Small*</small>
* <big>*Big*</big>
* <h1>*Header 1*</h1>
* <h2>*Header 2*</h2>
* <h3>*Header 3*</h3>
* <h4>*Header 4*</h4>
* <h5>*Header 5*</h5>
* <h6>*Header 6*</h6>
* <center>*Testo centrato*</center>
* <br /> Serve per andare a capo in maniera aggressiva.

### Elementi a Blocco:

Cioè che occupano il 100% della larghezza della pagina.

* Paragrafo.
* Divisore.
* Vari header.
* Lista ordinata.
* Lista non ordinata.

### Elementi In-Line:

Cioè che occupano solamente lo spazio necessario, in larghezza.

* Span.
* Bold.
* Corsivo.
* Small.
* Big.
* Collegamento ipertestuale.

### Attributi nativi di ogni Elemento:

* ID: è possibile specificare un ID per ogni elemento.
* Class: è possibile specificare l'appartenenza ad una classe di elementi.
* Style: è possibile ridefinire la grafica (lo stile) dell'elemento.
* Title: è possibile aggiungere una descrizione all'elemento, come un suggermento all'uso.

### Commenti:

I commenti sono possibili tramite la sintassi;

* <!-- Commenta qui -->

### Collegamenti Ipertestuali:

I collegamenti ipertestuli permettono di collegare una scritta ad una richiesta HTTP, come una ricerca sul web.

* <a> [attributi] > … </a>

**Attributi:**

* Href: in cui si definisce l'*URL* o il *percorso* del documento.
* Target: che specifica dove la nuova ricerca deve essere aperta, cioè se in una nuova scheda o nella scheda corrente.

*<a href="http://google.it" target="\_blank"> Click </a>*

###### *Ancore HTML:*

E' possibile scorrere in automatico una pagina attraverso dei collemanti ipertestuali, esempio:

*<a href="#fine\_pagina"> Click per andare a fine pagina </a>*

*…*

*… tutto il body*

*...*

*<p id="fine\_pagina"> Fine pagina. </p>*

### Immagini in HTML:

Il tag <img> è void e non ha chiusura.

* <img> [attributi] />

**Attributi:**

* Src: in cui si definisce l'*URL* o il *percorso* dell'immagine.
* Alt: che prevede un testo alternativo all'imagine nel caso in cui questa non possa essere renderizzata.
* Width e height: che specificano altezza e larghezza. Se viene modificato uno solo allora l'immagine verrà scalata.

### Liste in HTML:

Le liste sono elementi a Blocco, come ogni elemento della lista stessa.

Liste Ordinate: Liste non Ordinate:

*<ol> <ul>*

*<li>Primo</li> <li>ciao</li>*

*<li>Secondo</li> <li>come</li>*

*<li>Terzo</li> <li>stai?</li>*

*</ol>*  *</ul>*

### Tabelle in HTML:

Le tabelle vengono usate per inserire dati e in passato anche per costruire le strutture delle pagine web.

Si può usare l'attributo border nella table tag per mostrare o nascondere il bordo della tabella.

|  |  |  |
| --- | --- | --- |
| <tr> : Table Row. | <th> : Table Header. | <td> : Table Data. |

Esempio:

|  |
| --- |
| *<table border=1>*  *<tr>*  *<th>Username</th>*  *<th>Points</th>*  *</tr>*  *<tr>*  *<td>aremi</td>*  *<td>-999</td>*  *</tr>*  *…*  *</table>* |

###### *COLSPAN:*

Per rendere una cella larga più di una colonna posso usare l'attributo *colspan.*

Esempio:

<th colspan="2">Points</th>

###### *ROWSPAN:*

Per rendere una cella lunga più di una riga posso usare l'attributo *rowspan.*

Esempio:

<th rowspan="2">Points</th>

### HTML Divisor <div>:

Il tag <div> definisce una sezione in un documento HTML. Esso è un elemento a Blocco.

E' sato spesso insieme al CSS per costruire in layout di una pagina Web.

### HTML Forms:

Il tag *<form [Attributi]> … </form>* viene usato nell'HTML per definire delle forme, ovvero degli elementi che l'utente utilizzerà per prendere input. Gli input inseriti nelle *form* sono: caselle di testo, checkbox, bottoni ecc...

###### *Attributi delle form:*

* L'attributo *action* di una form va a definire l'azione che deve essere eseguita non appena viene rilevato l'inpt sulla form stessa. Spesso questo attributo viene usato per mandare dati ad una pagina web alla pressione di un pulsante.

Esempio:

*<form action="http://google.it/search">*

*...elementi form...*

*</form>*

* L'attributo *method* specifica invece al protocollo HTTP se deve effettuare una GET o una POST al rilevamento dell'inpt della form. Di default *method* è una GET.

Esempio:

*<form action="http://google.it/search" method="post">*

*...elementi form...*

*</form>*

**GET** viene usato per inviare dati chiaro, perciò non deve essere usato per inviare dati privati. Può inviare al massimo 255 char.

**POST** non mostra in chiaro i dati mandati e non ha limitazioni di dimensioni di upload.

### HTML Input:

L'elemento <input> permette all'utente di inserire dati, quali testo o numeri ecc... Ma anche di interagire con la pagine Web attraverso checkbox, bottoni ecc...

*<input type="{TIPO}" name="{NOME}" [ATTRIBUTI] />*

Ad ogni input deve essere associato un *name* che definisce la chiave da mandare alla form in caso di interazione dell'user. Se *name* viene omesso il dato dell'input non sarà mandato alla form.

###### *Input Testuali:*

*<input type="text" name="nickname" />*

*<input type="password" name="password" />*

|  |  |  |
| --- | --- | --- |
| *<input type="email" name="email" />* | *Il browser definisce un controllo sull'input.* | |
| *<input type="number" name="number" />* | | *In browser definisce un controllo sull'input.* | |

###### *Input – Attributi:*

* value: specifica un valore iniziale della casella input.
* readonly: *attributo void*, la casella input non può essere modificata.
* disabled: *attributo void*, la casella input è non-cliccabile e inutilizzabile.
* maxlength: lunghezza massima ammessa.
* placeholder: specifica un suggerimento che descrive la casella input.
* min e max: *tipo number*, specifica il valore massimo e minimo inseribile.

###### *Input – Checkbox:*

Sono i classici *checkbox* multiselezione. Il parametro *checked* seleziona per default la voce. Sono in-line.

###### *Input – Radio:*

Sono i checkbox a selezione singola... Stessi Attributi del *checkbox*.

### HTML TextArea:

E' un'area di testo multi-linea.

*<textarea name="textarea" rows="10" cols="50">*

*...scrivi...*

*</textarea>*

### HTML Select:

L'elemento *select* permette di implementare un menù di opzioni a scelta singola. La multiselezione è possibile attraverso l'attributo *void* multiple.

*<select name="select">*

*<option value="coca cola"> Coca Cola </option>*

*<option value="pepsi"> Pepsi </option>*

*</select>*

### HTML Button:

Il tag button definisce n bottono cliccabile. E' possibile inserire dentro questo anche testo e immagini.

*<button [Attributi] > Cliccami! :3 </button>*

# Il CSS

Il CSS è un linguaggio di stile che viene usato appunto per definire lo stile di elementi di testo, input, ecc… definiti in precedenza nella pagina HTML. Esso utilizza dei *selettori* per scorrere l'albero degli elementi HTML.

### CSS - Sintassi:

*[SELETTORI] {*

*{chiave html}:{regola CSS};*

*{…}:{…};*

*}*

*/\* Commenti qui \*/*

### CSS - Selettori:

Un *selettore* è usato per selezionare uno o più elementi HTML al fine di impostarne delle regole di stile.

Tipi di Selettori:

* Element Selector.
* Simple Selector.
* Attribute Selector.
* Pseudo-class Selector.
* Pseudo-element Selector.
* Combinators and Multiple Selectors.

###### *Element Selectors*

Seleziona tutti gli elementi HTML di uno stesso tipo:

*p { /\*seleziona tutti i <p>\*/*

*color: red;*

*}*

###### *Simple Selectors (Class)*

Si denota con il ' . ' seguito dal nome della classe:

*.class {*

*…*

*}*

###### *Simple Selectors (ID)*

Si denota con il ' # ' seguito dal nome della ID:

*#id{*

*…*

*}*

###### *Simple Selectors (Universale)*

*Si denota con il ' \* ':*

*\* {*

*…*

*}*

###### *Combinatori*

I combinatori sono degli operatori che permettono di combinare efficacemente più *selectors* fra loro. Sono molteplici e si denotano con;

* **Selettore di Discendenza**: ' *spazio* ' permette di selezionare un elemento annidato dentro un altro elemento.
* **Selettore di Figlio**: ' > ' permette di selezionare il figlio diretto di un elemento.
* **Selettore sibling adiacente**: ' + ' permette di selezionare un elemento che è fratello diretto di dell'elemento scelto.
* **Selettore sibling generale**: ' ~ ' permette di selezionare ogni elemento che fratello non diretto dell'elemento scelto (cioè allo stessa altezza nell'albero HTML).

Testo alternativo generato dal computer:
Heading 1 
Heading 2 Testo alternativo generato dal computer:
HTML 
< p»paragraph p» 
< p»paragraph 4</p» 
Lpx 
olor, 
dashed black; 

  <= Output

###### *Multiple Selectors*

I selettori possono essere scritti a cascata, separati da ' , ' per applicare le stesse regole a più elementi contemporaneamente:

*p, h2 {*

*font-size: 1.6em;*

*}*

###### *Attribute Selectors*

*Si denota con il ' [] ':*

|  |  |  |
| --- | --- | --- |
| *[attribute] {* |  | *[attr=value] {* |
| *…* |  | *…* |
| *}* |  | *}* |

##### Gli Attributi Custom in HTML:

Gli attributi possono anche essere dall'utente…

*<p class="albero" disabled cicoria="affettata"> … </p>*

**disabled** è un attributo nativo dell'elemento <p>

**cicoria** è invece un attributo custom che ha valore "**affettata**"

###### *Pseudo-class Selectors*

Una *pseudo-classe* in CSS è uno stato in cui un elemento HTML può trovarsi a causa dell'interazione dell'utente con l'elemento stesso. Ad esempio un pulsante puntato dal cursore del mouse si troverà nello stato :hover fino a quando non sarà più puntato.

Gli stati sono molteplici e si denotano con il nome dell'elemento di ci voglio controllare lo stato, il ' : ' seguito dal nome dello stato:

*a:visited{ /\*controlla se un link ipertestuale è stato visitato\*/*

*…*

*}*

###### *Pseudo-element Selectors*

Uno pseudo-elemento è un elemento "fantasma" per selezionare una determinata parte dell'elemento scelto. Si denotano con il ' :: ' e anch'essi sono molteplici.

*[href^=http]::after { /\*aggiunge quel carattere speciale dopo il link\*/*

*content: 'Ө' ;*

*}*

### CSS - Rules:

* color: red, rgb(x,x,x), #hex ;
* text-decoration: none, underline red ;
* text-transform: capitalize, uppercase, lowercase, none, full-width ;
* text-shadow: x-px, y-px, blur-px, color ;
* text-align: left, right, center, justify ;
* line-height: normal, px, n% ;
* letter-spacing: normal, px ;
* font-size: smaller, larger, px ;
* font-weight: normal, bold, lighter, bolder, n ;
* font-family: serif, sans-serif, monospace, cursive ;
* border: width style color ;
* border-top: width style ;
* border-right: width style ;
* border-left: width style ;
* border-bottom: width style ;
* margin: top right bottom left ;
* padding: top right bottom left ;



* display: block, inline, none, inline-block ;
* position:
  + *relative*, è un elemento la cui posizione è calcolata relativamente a se stesso, ovvero ai parametri che gli diamo noi e allo spazio che esso occupa.
  + *absolute*, è un elemento la cui posizione è calcolata in funzione dell'elemento padre.
  + *fixed*, è un particolare *absolute* in cui però la posizione viene calcolata a "run-time", permettendo quindi una adattabilità alle variazioni del padre.
  + *sticky*, è un elemento la cui posizione è fissata, non dinamica.

position: static, relative, absolute, fixed, sticky ;

* top: px ;
* right: px ;
* bottom: px ;
* left: px ;

* float: left, right, none ; //Serve per dire ad un contenuto di posizionarsi in una certa maniera rispetto al suo contenitore.
* clear: left, right, none ; //Serve per pulire la <float>
* clearfix //bho
* background:
  + background: color ;
  + background: url("test.jpg") repeat-y ;
  + background: border-box red ;
  + background: no-repeat center 80% url("test.jpg")
* background-position: top, bottom, left, right, center ;
* background-size: cover, contain ;
* cursor: pointer, auto ;
* overflow: visible, hidden ; //Gestisce cosa fare quando il contenuto esce dal contenitore.

### CSS - Importare File:

Per importare un file **CSS** è necessario specificare nell' *HEAD* del file *HTML* il link al file stylesheet in formato \*.css

*<link rel="stylesheet" href="…/assets/mystyles.css" media="all" />*

## !important

A prescindere da qualsiasi regola disposta in un qualsiasi ordine applicata ad un certo elemento, devi applicare la regola preceduta dal *!important*

Viene usato quindi per forzare una regola in CSS.

*.class {*

*display: none !important;*

*}*

### Classi Ausiliarie:

E' consigliabile scrivere delle classi generiche con regole generiche e singole, in modo da poterle applicare a più oggetti html.

*.absolute {*

*position: absolute;*

*}*

## CSS Responsive

Un problema comune, nell'HTML e nel CSS consiste nello sviluppare la pagina web in modo tale che questa si adatti alle tantissime risoluzioni dei dispositivi che accedono alla pagina stessa.

Testo alternativo generato dal computer:
The basic idea is to organize 
the various components of 
the web page through a new 
unit of measure, this is 
called column. 
A web element (from body to 
any Other) has a total width 
of 12 columns. 
TWO THIRD 
IO Column 
THIRD Per fare ciò si è pensato di implementare una sorta di tabella immaginaria di *<div>* composta da 12 Colonne e N righe.

Una implementazione di tale tabella, per pc fisso, sarebbe:

Testo alternativo generato dal computer:
box-gizing: 
Content: " 
Clear: 
display: black: 
float: 
padding' 15px: 
COI- 1 (Width _ 
833%:) 
031-2 (width: 
031-3 ( Width: 
3333%:} 
COI. S (width_ 
031-6 (width: 
7 (width_ 
- 5833%:} 
031- g (width_ 
(width: 
col. 10 {Width: 83 33%,) 
col-12 {width: 100%:} 

In questo modo, attraverso le Media Query, possiamo modificare la larghezza di ogni colonna in base alla risoluzione del dispositivo utente:

@media screen and (min-width: 900px) {

.col-1 {

width: 100%

}

}

Quindi per mobile si ha:

Testo alternativo generato dal computer:
So add the support for mobile devices 
/• desktop: 
g. JAN;) 
"idth: 

# Basi di Javascript

## Informazioni base:

JavaScript è un linguaggio di programmazione considerato come il terzo layer nella programmazione WEB ed è un linguaggio di programmazione **Client-Side**.

Si tratta di un linguaggio interpretato, quindi non è applicata una compilazione del codice, ciò implica che dei possibili errori di sintassi non saranno segnalati ed il codice sarà eseguito ugualmente anche in presenza di tali errori.

E' un linguaggio orientato agli oggetti e, nonostante sia tipizzato, non distingue i tipi delle variabili. Le variabili possono anche non essere dichiarate (dynamic typing).

Il codice Javascript viene eseguito dal browser engine (interprete) dopo aver caricato tutto l'HTML ed il CSS.

Perciò è possibile modificare dinamicamente l' HTML ed il CSS attraverso il codice.

L'interprete JavaScript prima di eseguire il codice riga per riga, legge tutto il file e carica in memoria le variabili **var** e le **funzioni**.

###### *Riflessività del linguaggio:*

Essendo un linguaggio interpretato il codice viene eseguito dall'alto verso il basso e ci permette di utilizzare la riflessione, ovvero quella proprietà che ci permette di far variare il comportamento, la struttura, la funzionalità stessa di un oggetto, direttamente a run-time.

### Sicurezza:

Il browser esegue tutto il codice Web all'interno di una sandbox, ovvero il browser non fornisce ad esempio le API per la lettura e scrittura su file, in questo modo si impedisce di accedere all'Hard-Disk di un qualsiasi utente che naviga in una pagina Web che esegue del codice JavaScript.

Le estensioni del browser invece permettono di eseguire del codice fuori dalla sandbox perché le estensioni stesse sono fuori dalla sandbox, per tanto bisogna assicurarsi della sicurezza di tali strumenti, perché potenzialmente pericolosi per l'utente.

### Dove e come aggiungere il codice JavaScript:

Aggiungere il codice con il tag:

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

All'interno del body, ma alla fine di tutti gli elementi. In questo modo forziamo il browser a caricare il codice come elemento finale.

## Sintassi: regole

### Hello Word:

alert('Hello world!');

console.log('Hello world!');

* + *alert* è una operazione bloccante, cioè blocca il flusso di esecuzione del codice.
  + *console* è un oggetto fornito da una API del browser che ci permette di interagire con la console del browser.

### Commenti:

// commento in-line

/\*\*/ commento multilinea

### Variabili:

Le variabili possono cominciare anche con '\_' e '$'. Per convenzione le variabili che cominciano con '$' indicano variabili che contengono ***DOM (ovvero elementi HTML)***.

###### *Dichiarazione:*

Ci sono 3 modi per dichiarare una varabile:

* + Con la parola chiave **var**.  
    Ad esempio: *var x = 42*; In questo modo si dichiara una variabile globale.
  + Con la parola chiave **let**.  
    Ad esempio *let y = 13*; In questo modo si dichiara una normale variabile locale.
  + Con la parola chiave **const**.  
    Ad esempio: *const y = 13*; In questo modo si dichiara una normale variabile locale costante.

### Tipi primitivi delle variabili:

Il tipo fondamentale su cui si basa tutto JavaScript è la **stringa**. Nel caso peggiore (ad esempio *2=="2"*) JavaScript trasforma implicitamente in stringa.

* + **string**

var albero= 'Hello Kitty';

var \_pippo= "Pamela's Place";

console.log(albero[0]); // 'a'

console.log(\_pippo.length); // 14

* + **number**

var myAge = 28;

var pi = 3.14;

* + **boolean**

var catsAreBest = true;

var dogsRule = false;

* + **undefined**

var notDefinedYet;

* + **null**

var goodPickupLines = null;

* + **object and functions**

### Conversione tipi delle variabili:

Le variabili possono cambiare tipo dinamicamente.

var x;

x = 2;

x = 'Hi';

console.log(typeof(x));

### Costrutto IF: uguale al C++.

### Costrutto Switch: uguale al C++ ma in più i case possono gestire anche stringhe, oltre gli interi.

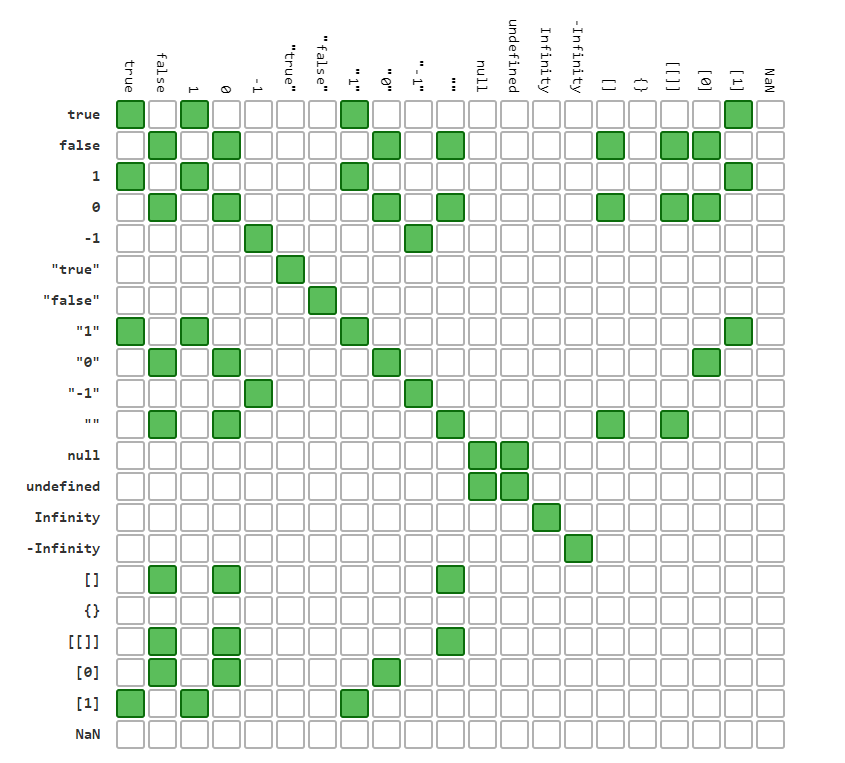
### Espressioni condizionali:

|  |  |
| --- | --- |
| **Argument Type** | **Result** |
| Undefined | false |
| Null | false |
| Boolean | The result equals the input argument (no conversion). |
| Number | The result is false if the argument is +0, −0, or NaN; otherwise the result is true. |
| String | The result is false if the argument is the empty String (its length is zero); otherwise the result is true. |
| Object | true |

### Operatori condizionali:

|  |  |
| --- | --- |
| **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 |

### Tabella di verità:



### Costrutto WHILE: uguale al C++.

### Costrutto FOR: uguale al C++.

### Costrutto FOR-IN, oggetti Iterable:

Il for-in crea dei cicli in cui itera gli indici di un oggetto Iterable (un array, ad esempio):

for(var index in arr) {

  console.log(arr[index]);

  // more statements

}

### Costrutto FOR-OF, oggetti Iterable:

Il for-of crea dei cicli in cui itera gli **elementi** che sono presenti dentro un oggetto Iterable (un array, ad esempio):

for(var obj of arr) {

  console.log(obj);

  // more statements

}

## Array:

var fruits = ["Apple", "Banana"];

console.log(fruits.length); // 2

var first = fruits[0]; // Apple

var last = fruits[fruits.length - 1]; // Banana

* + **Un array è un oggetto e come tale dispone di funzioni (***push()***) e parametri (***length***)**:

###### *forEach():*

E' un metodo che prende in input una funzione che verrà eseguita a tutti gli elementi dell'array.

array.forEach(function(item, index, array)){

  console.log(item, index);

});

// Apple 0

// Banana 1

###### *push():*

var newLength = fruits.push("Orange"); // ["Apple", "Banana", "Orange"]

###### *pop():*

var last = fruits.pop(); // remove Orange (from the end) // ["Apple", "Banana"];

###### *shift():*

var first = fruits.shift(); // remove Apple from the front // ["Banana"];

###### *unshift():*

var newLength = fruits.unshift("Strawberry") // add to the front // ["Strawberry", "Banana"];

###### *indexOf():*

var pos = fruits.indexOf("Banana"); // 1

###### *splice():*

var removedItem = fruits.splice(pos, 1); // this is how to remove an item // ["Strawberry", "Mango"]

###### *slice():*

var shallowCopy = fruits.slice(); // this is how to make a copy // ["Strawberry", "Mango"]

## Funzioni:

In questa maniera definiamo una funziona globale:

function square(number) {

  return number \* number;

}

In questo modo invece definiamo funzione, che può essere anonima (senza un nome), che attribuiamo ad una variabile:

var square = function(number) { return number \* number; };

var x = square(4); // x gets the value 16

**Questo aspetto è molto importante perché ci permette di ridefinire il comportamento di una funzione, anche a run-time.**

E' quindi possibile dichiarare delle funzioni normali funzioni locali attribuendole a delle variabili **let**.

Di conseguenza è anche possibile passare delle funzioni come parametri:

function map(f, a) {

  var result = []; // Create a new Array

  var i; // Declare variable

  for (i = 0; i != a.length; i++)

    result[i] = f(a[i]);

      return result;

}

var f = function(x) {

  return x \* x \* x;

}

var numbers = [0,1, 2, 5,10];

var cube = map(f,numbers);

console.log(cube);

La ricorsione è ammessa.

### Funzioni Arrow (Arrow Functions) - Funzioni Lambda:

Ogni metodologia di definizione di una funzione spiegata in precedenza ha la caratteristica di **definire nella funziona un proprio THIS che viene inizializzato in base a come la funzione viene chiamata.**

Una funzione *Arrow* invece non possiede una proprio THIS.

Questo tipo di dichiarazione viene usato principalmente per il *main* del programma, per non permettere alla console del browser di rivelare il corpo del *main* stesso (p. Scope e Closure).

E viene usato anche per definire i metodi degli oggetti.

(param1, param2, …, paramN) => { statements }

(param1, param2, …, paramN) => expression

// equivalent to: => { return expression; }

// The parameter list for a function with no parameters should be written with a pair of parentheses.

() => { statements }

## Scope e Closure:

Le variabili possiedono un loro livello di scope, sono cioè visibili nella funzione in cui sono definite.

Le Closure sono una delle funzionalità più potenti di JavaScript. JavaScript consente infatti l'annidamento di funzioni e garantisce alla funzione interna pieno accesso a tutte le variabili globali e funzioni globali definite all'interno della funzione esterna (e tutte le altre variabili e funzioni a cui la funzione esterna ha accesso).

Esempio di Closure di una funzione:

var pet = function(name) {   // La funzione esterna prende una variabile chiamata "name".

var s1 = "Mi "; //La funzione interna getName HA accesso alla variabile *var.*

let s2 = "chiamo "; //La funzione interna getName NON ha accesso alla variabile *let.*

var getName = function(s2) {

    return s1+s2+name;             // La funzione interna ha accesso alla variabile "name" della funzione esterna.

}

return getName;    // Ritorna la funzione getName, esponendola allo scope esterno alla funzione esterna.

}

var myPet = pet('Vivie');

myPet();                     // Ritorna "Mi chiamo Vivie"

Esempio di Closure del main:

() => {

…

corpo della funzione main

…

}()

Le due parentesi finali servono per eseguire la funzione stessa.

## JSON *(JavaScript Object Notation)*:

JSON è un formato di rappresentazione dei dati sottoforma di stringa.

Rappresentare un oggetto a stringa è utile perché il *client* ed il *server* possono essere sviluppati con linguaggi di programmazione differenti, serve quindi un qualcosa, comune ad entrambi, per passare le informazioni e far comunicare le due parti.

JSON è fondamentalmente una rappresentazione degli oggetti JavaScript sottoforma di stringhe.

Nel JSON, essendo un formato molto comune e usato, il *parsing* avviene attraverso delle funzioni *parser*, implementate nei vari linguaggi. Tali funzioni sono native nel JavaScript.

## Oggetti:

Un oggetto è una collezione di dati e/o funzionalità correlati. La creazione di oggetto comincia con la definizione e inizializzazione di una variabile JS.

var obj= { [key]:[value][,...]};

var person = {

  name: [Federico, Fausto],

   age: 28,

  gender: 'male',

  interests: ['nintendo', 'cloud'],

 bio: function() {

   alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years old. He likes ' + this.interests[0] + ' and ' + this.interests[1] + '.');

},

   greeting: function() {

    alert('Hi! I\'m ' + this.name[0] + '.');

  }

};

Per accedere ad un parametro o per utilizzare una funzione di un oggetto si può usare la notazione del punto o la notazione delle parentesi quadre.

### 

### Notazione delle parentesi quadre:

E' possibile accedere ai parametri o funzioni di un oggetto con la notazione:

person['age'] = …

person['bio'] = …

delete person.age;

### 

###### *Array come oggetti:*

Abbiamo già visto che un array, in JS, è un oggetto con i suoi parametri (*length*) e le sue funzioni (*push()*).

E' possibile **aggiungere** nuovi parametri o funzioni agli oggetti ed è anche possibile riscrivere le funzioni native, a run-time…

var mioArray = ["Gabriele"]; // Il parametro mioArray.length vale 1.

mioArray["Cognome"] = ["Cipolla"]; // Creo un nuovo parametro ==> mioArray.Cognome = "Cipolla" e mioArray.length vale 1.

// Creo un nuovo parametro che inizializzo come la funzione pop(). ==> var togliPrimoElemento = this.pop()

mioArray["togliPrimoElemento"] = mioArray.pop();

mioArray.togliPrimoElemento();

mioArray.push = (obj) => { // Riscrivo la funzione push di mioArray.

    mioArray.unshift(obj);

}

E' anche possibile **rimuovere** parametri o funzioni agli oggetti, a run-time…

delete mioArray.Cognome;

delete mioArray.push();

### this:

la parola chiave ***this*** rappresenta l'oggetto stesso.

## Classi:

E' possibile implementare delle classi:

class Student {

name;

surname;

#gender; //private

constructor(name, surname) {

this.name = name;

this.surname = surname;

}

}

## DOM *(Document Object Model)*:

Il DOM è una interfaccia per il documento *HTML* che ci permette di accedere all'*HTML* attraverso una serie di funzioni contenute nella nostra variabile nativa ***document***.

Esistono infatti 3 variabili che sono già predisposte dal nostro browser:

* + ***window***: rappresenta il browser stesso.
  + ***document***: è l'oggetto che rappresenta tutto il nostro documento *HTML*.
  + ***console***: è l'oggetto che gestisce la console del browser.

In questo modo il nostro linguaggio di programmazione può connettersi con la pagina WEB.

Per esempio lo standard DOM specifica un metodo ***getElementByTagName("tag")*** che ritorna un array di tutti gli elementi di quel <tag> del documento *HTML*.

var paragraphs = document.getElementsByTagName("DIV"); // L'HTML è case Insensitive!

###### *Modificare l'HTML a run-time:*

Il DOM di JavaScript presenta delle funzioni per modificare a run-time i nostri elementi HTML e anche il loro stile. Si tratta di funzioni native che si presentano molto complicate nell'uso, se il file HTML o CSS è complesso, per questo motivo è consigliato l'uso del FrameWork **JQuery** per effettuare questo genere di operazioni:

* + var heading = document.createElement("h1");
  + var heading\_text = document.createTextNode("Big Head!");
  + heading.appendChild(heading\_text);
  + document.body.appendChild(heading);
  + …

### Come e quando si accede al DOM? Evento *onload()*

Il DOM sarà reso disponibile nel momento in cui la pagina sarà resa disponibile dal browser, ovvero quando il caricamento del codice *HTML* sarà ultimato.

Per conoscere il momento esatto in cui il codice *HTML* è stato caricato è possibile usare l'evento ***onload*** all'interno del body:

* + <body onload="window.alert('Page loaded!');">
  + window.onload = function() {

alert("Benvenuto")

};

Questo evento, contenuto nell'oggetto window, viene scatenato dal browser quando il documento sarà caricato.

Il problema di questa funzione è che non considera il caricamento di tutte le immagini ed altri elementi dentro il nostro documento, per ovviare a questo problema utilizzeremo un'altra funzione del framework **JQuery**.

###### *Alcuni metodi dell'interfaccia DOM:*

* + document.getElementById(id)
  + document.getElementsByTagName(name)
  + document.getElementsByClass(className)
  + document.createElement(name)
  + parentNode.appendChild(node)
  + element.innerHTML
  + element.style.left
  + element.setAttribute()
  + element.getAttribute()
  + element.addEventListener()
  + window.content
  + window.onload
  + window.content
  + window.onload
  + console.log()
  + window.scrollTo()

## Eventi DOM:

Vi sono 2 modi consigliati per gestire gli eventi dell'interfaccia DOM in JS:

* + Utilizzando la funzione addEventListener:

// Assuming myButton is a button element

myButton.addEventListener('click', greet);

function greet(event){

// print and have a look at the event object

// always print arguments in case of overlooking any other arguments

console.log('greet:', arguments);

alert('hello world');

}

* + Sovrascrivendo la funzione stessa dell'elemento.  
    Il problema di questo metodo è che potrà essere settato soltanto quell'evento a quell'elemento:

// Assuming myButton is a button element

myButton.onclick = function(event){alert('Hello world');};

###### *addEventListener():*

Il metodo element.addEventListener('event', (event) => {…}) prende in input un type di evento ed una funzione lambda di callback.

Tale funzione di callback prenderà in input due parametri;

* + *this*, ovvero l'elemento chiamante (implicito).
  + event, ovvero l'evento stesso.

Avendo il riferimento all'evento stesso allora sarà possibile fermare evento, cioè impedire l'esecuzione di default dell'evento, attraverso il metodo event.preventDefault().

###### *preventDefault():*

Il metodo event.preventDefault(), nella sovrascrittura di un evento nativo, comunica al Browser di non eseguire il codice nativo dell'evento ma di eseguire esclusivamente il codice scritto da noi.

Questa cosa è comoda quando, ad esempio in una *form*, si vuole controllare se l'email inserita nel campo è uguale a quella inserita nel campo conferma email.

###### *setTimeout():*

Questa funzione non necessita di nessun oggetto chiamante (usala come se fosse un metodo statico) e prende come parametri:

* + Una funzione lambda di callback.
    - setTimeout(() => {

…

}, 3000)

* + un tempo in millisecondi.

# JQuery

## Informazioni generali

JQuery è un Framework di JavaScript creato allo scopo di accedere in maniera pratica all'interfaccia DOM.

Jquery è infatti una libreria che serve a manipolare e gestire il documento HTML in maniera molto più comoda rispetto al JS nativo.

Per importare la librearia JQuery è necessario scaricare la libreria JS ed importarla tramite il tag <script> nell'HTML.

JQuery usa una sintassi con questo formato $(selector).action() dove:

* *$* è il nome dell'oggetto di tipo JQuery, che a sua volta è di tipo JS.
* *(selector)* può essere sia una stringa di **Selettori CSS**, sia un oggetto DOM, ad esempio **document**.
* *action()* corrisponde all'evento da triggerare.

###### *.onload() vs .ready():*

Ricorderai che l'evento *onload* dell'oggetto *window* ci segnala il caricamento del nostro documento *HTML*…

Il problema di questa funzione è che non considera il caricamento di tutte le immagini ed altri elementi dentro il nostro documento.

La funzione .ready() che ci fornisce la libreria JQuery invece verrà eseguita solamente se TUTTO il nostro documento verrà caricato.

### Esempi di Selettori CSS:

Il concetto di base del JQuery è *"prendi uno o più oggetti e fai qualcosa"*

* $( "#myId" ); *// Nota che l'ID deve essere unico.*
* $( ".myClass" );
* $( "input[name='first\_name']" );
* $( "#contents ul.people li" );
* $( "div.myClass, ul.people" );
* $( "a.external:first" );
* $( "tr:odd" );

*// Select all input-like elements in a form (more on this below).*

* $( "#myForm :input" );
* $( "div:visible" );

*// All except the first three divs,*

* $( "div:gt(2)" );

Per comodità è possibile conservare gli elementi selezionati tramite JQuery in delle variabili JS:

**var** divs = $( "div" );

Inoltre è possibile rifinire e filtrare i selettori attraverso apposite funzioni:

*// Refining selections.*

* $( "div.foo" ).has( "p" ); *// div.foo elements that contain <p> tags*
* $( "h1" ).not( ".bar" );  *// h1 elements that don't have a class of bar*
* $( "ul li" ).filter( ".current" ); *// unordered list items with class of current*
* $( "ul li" ).first();              *// just the first unordered list item*
* $( "ul li" ).eq( 5 );              *// the sixth*

## Modificare l'HTML a run-time

### Manipolare gli elementi JQuery, funzioni:

JQuery offre moltissime funzioni get e set per manipolare gli oggetti, eccone alcune:

* **.html()** – Get or set the HTML contents.
* **.text()** – Get or set the text contents; l'HTML verrà rimosso.
* **.attr()** – Get or set the value dell'elemento che possiede l'attributo generico fornito.
* **.width()** – Get or set the width in pixels of the first element in the selection as an integer.
* **.height()** – Get or set the height in pixels of the first element in the selection as an integer.
* **.position()** – Get an object with position information for the first element in the selection, relative to its first positioned ancestor. *This is a getter only*.
* **.val()** – Get or set the value of form elements.

*// Esempio: Changing the HTML of an element.*

* $( "#myDiv p:first" ).html( "New <strong>first</strong> paragraph!" );

### Inserimento prima o dopo un dato elemento:

Ci sono generalmente due approcci per posizionarne nuovi elementi del DOM:

1. Posizionare gli elementi selezionati in relazione a un altro elemento.
2. Posizionare un elemento in relazione agli elementi selezionati.

Il metodo *.insertAfter()* posiziona gli elementi selezionati dopo l'elemento fornito come argomento.

Il metodo *.after()* inserisce l'elemento fornito come argomento dopo l'elemento selezionato.

Diversi altri metodi seguono questo modello: *.insertBefore()* e *.before()*, *.appendTo()* e *.append()*, *.prependTo()* e *.prepend()*.

*// Moving elements using different approaches.*

*// Make the first list item the last list item:*

**var** li = $( "#myList li:first" ).appendTo( "#myList" );

*// Another approach to the same problem:*

$( "#myList" ).append( $( "#myList li:first" ) );

*// Note that there's no way to access the list item that we moved, as this returns the list itself.*

### Rimuovere elementi:

Esistono due modi per rimuovere elementi dalla pagina: *.remove()* e *.detach()*.

* *.remove()* : si utilizza quando si desidera rimuovere definitivamente l'elemento dalla pagina. Questa funzione ritorna gli elementi rimossi privati dei loro dati e degli eventi a cui erano attaccati.
* *.detach()* : si utilizza quando si desidera rimuovere temporaneamente un elemento che poi si vorrà ripristinare nella sua interezza. Infatti questa funzione ritorna gli elementi rimossi conservando sia i loro dati che gli eventi a cui erano attaccati.

### Creare elementi nuovi:

*// Creating new elements from an HTML string.*

**var** prova = $( "<p>This is a new paragraph</p>" );

**var** prova = $( "<li class=\"new\">new list item</li>" );

*// Creating a new element with an attribute object.*

**var** prova = $( "<a/>", {html: "This is a <strong>new</strong> link", "class": "new", href: "foo.html"});

### Manipolazione Attributi:

JQuery offre la possibilità di aggiungere e modificare degli attributi, anche custom, degli elementi HTML.

Attraverso i due metodi .attr() e .data() :

* $( "Selector" )**.attr**({ href: "john-wick-3.html", banana: "arancione" }); // Modifica di più attributi.  
  $( "#myDiv a:first" )**.attr**( "href", "john-wick-3.html" ); // Modifica di un solo attributo.  
  Attraverso questo metodo è possibile modificare e aggiungere uno o più attributi ad un selettore
* L'altro metodo permette di modificare o aggiungere tutti i parametri del tipo ***data-parameter*** quindi è più limitata.

### Attraversamento ad Albero:

Gli elementi HTML sono gestiti con una struttura ad albero. Il JQuery ci mette a disposizione degli ulteriori metodi di attraversamento che approfittano proprio di questa struttura (ognuno di questi metodi può opzionalmente accettare una stringa di Selector):

*// Selecting an element's direct parent:*

* $( "span.subchild" ).parent();

*// Selecting all the parents of an element that match a given selector:*

* $( "span.subchild" ).parents( "div.parent" );
* $( "span.subchild" ).parents();

*// Selecting all the parents of an element up to, but \*not including\* the selector:*

* $( "span.subchild" ).parentsUntil( "div.grandparent" );

*// Selecting the closest parent, note that only one parent will be selected*

*// and that the initial element itself is included in the search:*

* $( "span.subchild" ).closest( "div" );

*// returns [ div.child ] as the selector is also included in the search:*

* $( "div.child" ).closest( "div" );

*// Selecting an element's direct children:*

* $( "div.grandparent" ).children( "div" );

*// Finding all elements within a selection that match the selector:*

* $( "div.grandparent" ).find( "div" );

## Async JS e Ajax

Il *client* comunica con il *server* attraverso richieste (*get* o *post*). Queste richieste vengono eseguite tramite delle chiamate, delle funzioni, che si distinguono in: **chiamate sincrone** e **chiamate asincrone**.

* Una chiamata si definisce **sincrona** quando il *client* che effettua la richiesta rimane bloccato in attesa di risposta.  
  Se il *server* in quel momento è impegnato con un'altra richiesta, allora il *client* rimarrà bloccato fin quando il *server* non gestirà la sua richiesta. Questo tipo di chiamata si effettua quando ci si aspetta dal *server* una risposta in tempo reale, cioè quando ci si aspetta che il *server* sia sicuramente disponibile.
* Una chiamata si definisce **asincrona** quando il *client* che effettua una richiesta non sa quando quest'ultima verrà soddisfatta dal *server*. Quindi è ovvio che il *client* non può bloccare la sua esecuzione in attesa di una risposta, infatti con questo tipo di chiamata il resto del codice continuerà ad essere eseguito permettendo di poter continuare a svolgere attività.  
  Quando il *server* manderà la risposta allora verrà triggerata, nel client, una **funzione di callback** per gestire la risposta.

Per la navigazione sul WEB i client effettuano chiamate asincrone, ovviamente.

**Da adesso, per via delle chiamate asincrone, bisognerà ragione in funzione del tempo!**

### Metodo .get():

Nativamente JS mette a disposizione una classe (*XMLHttpRequest*) che definisce dei metodi per gestire queste chiamate asincrone. Il problema di questa classe è che viene gestita diversamente in base al browser che si utilizza, in più è anche complicata da usare.

Anche in questo campo ci viene incontro il framework JQuery, che permette di chiamare sull'oggetto JQuery ' **$** ' dei metodi che semplificano l'esecuzione di una chiamata asincrona e la gestione di una possibile risposta o di possibili errori.

$.get( "url-to-request.php", **callback-function**(risposta) {…}

###### *Esempio di get che non considera il tempo di risposta:*

**var** response;

$.get( "request.php", **function**( r ) {

   response = r;

});

console.log( response ); *// undefined*

###### *Esempio di get che considera il tempo di risposta:*

$.get( "request.php", **function**( response ) {

   console.log( response ); *// server response*

});

### Metodo .post():

Il metodo *post* prende una *url*, un *body* e una funzione di *callback*:

$.post("url", {chiave : "valore"}, () => {});

### Metodo .ajax():

Il metodo *get* ed il metodo *post* sono in realtà contenti all'interno di un metodo più grande chiama ***ajax***.

Questo metodo ci permette di definire in maniera più completa anche altre impostazione di una chiamata,

prende infatti come parametri:

$.ajax({ url: "post.php", data: { id: 123 }, type: "GET", dataType : "json" })

// The URL for the request

// The data to send (will be converted to a query string)

// Whether this is a POST or GET request

// The type of data we expect back

All'interno di questa funzione è possibile prevedere delle callback di **.done()**, **.fail()** e .**always()** per gestire la risposta del server.

Per gestire le richieste è consigliato usare questo metodo, piuttosto che il *get* o *post*.

Comunque è meglio approfondire i *parametri* e le *callback* su internet:

<https://api.jquery.com/jQuery.ajax/>

## JSON:

Il JSON è un modo per rappresentare oggetti JS sotto forma di stringa, ed è comodo nello scambio di dati tra un *client* ed un *server*. I due infatti utilizzeranno linguaggi diversi e serve qualcosa di comune per permettere la comunicazione e lo scambio dei dati.

Per trasformare un oggetto JS in una stringa JSON si utilizza una interfaccia chiamata appunto JSON che mette a disposizione due metodi:

* *stringify(obj)*, che trasforma un oggetto JS in una stringa JSON.
* *parse(obj)*, che trasforma una stringa JSON in un oggetto JS.

# Basi di PHP

### Qualche info:

E' un linguaggio interpretato usato nella programmazione Web lato *Server*. Quindi gli script PHP sono eseguiti nel *server*.

Permette di creare pagine Web dinamiche, ovvero pagine web che rispecchiano il comportamento dell'ambiente circostante (ad esempio le pagine Web possono essere collegate ad un database, se cambia il database cambia anche la pagina).

**Guida al PHP:**

<https://www.w3schools.com/php/>

Il *Server* interpreta il **PHP**, formalizza l'**HTML** e lo restituisce come risposta al *Client*.

Differenza tra " " e ' ' :

$txt = "DMI"

Le doppie virgolette interpretano le variabili presenti nella stringa stessa; "I love $txt" => I love DMI

Gli apici non interpretano le variabili; 'I love $txt' => I love $txt

### Sintassi:

Uno script PHP può essere posizionato ovunque all'interno del documento e deve essere contenuto nei tag:

|  |
| --- |
| <?php  // code goes here...  ?> |
| <?= //expression php here… ?> | |

Le key-words sono case-insensitive mentre le variabili sono case-sensitive.

### Commenti:

I commenti sono uguali al C, con in più la possibilità di commentare una riga anteponendo '#'.

### Variabili:

PHP è debolmente tipizzato, come JavaScript… perciò si usano le stesse regole del JS con la differenza che tutte le variabili devono essere dichiarate anteponendo il '$' davanti il nome della variabile.

|  |
| --- |
| <?php  $txt = "Hello world!";  $x = 5;  $y = 10.5;  ?> |

### Echo:

L'istruzione *echo* stampa a schermo:

### Scope:

Le variabili hanno lo stesso scope del C, in più è possibile dichiarare delle variabili static sono locali allo scope non vengono eliminate dalla memoria.

|  |
| --- |
| <?php  function myTest() {    static $x = 0;    echo $x;    $x++;  }  ?> |

### Data Type:

PHP supporta i data type:

* String
* Integer
* Float (floating point numbers - also called double)
* Boolean
* Array
* Object
* NULL
* Resource

*var\_dump($x)*; //Ritorna il tipo della variabile.

### Tipo String:

* *strlen(*) function returns the length of a string.
* *str\_word\_count()* function counts the number of words in a string.
* *strrev()* function reverses a string.

…

### Variabili Costanti:

L'istruzione *define(nome, valore, case-sensitive)* definisce una variabile costante.

**Le variabili e array costanti non hanno bisogno del '$'.**

* *name*: Specifies the name of the constant
* *value*: Specifies the value of the constant
* *case-insensitive*: Specifies whether the constant name should be case-insensitive. Default is false

### Operatori Aritmetici:

|  |  |
| --- | --- |
| +        Addition | Sum of $x and $y |
| -        Subtraction | Difference of $x and $y |
| \*        Multiplication | Product of $x and $y |
| /        Division | Quotient of $x and $y |
| %        Modulus | Remainder of $x divided by $y |
| \*\*        Exponentiation | Result of raising $x to the $y'th power |

### Operatori di Assegnamento:

|  |  |
| --- | --- |
| x = y | The left operand gets set to the value of the expression on the right |
| x += y | Addition |
| x -= y | Subtraction |
| x \*= y | Multiplication |
| x /= y | Division |
| x %= y | Modulus |

### Operatori di Confronto:

|  |  |
| --- | --- |
| ==        Equal | Returns true if $x is equal to $y |
| ===        Identical | Returns true if $x is equal to $y, and they are of the same type |
| !=        Not equal | Returns true if $x is not equal to $y |
| <>        Not equal | Returns true if $x is not equal to $y |
| !==        Not identical | Returns true if $x is not equal to $y, or they are not of the same type |
| >        Greater than | Returns true if $x is greater than $y |
| <        Less than | Returns true if $x is less than $y |
| >=        Greater than or equal to | Returns true if $x is greater than or equal to $y |
| <=        Less than or equal to | Returns true if $x is less than or equal to $y |
| <=>        Spaceship | Returns an integer less than, equal to, or greater than zero, depending on if $x is less than, equal to, or greater than $y. Introduced in PHP 7. |

### Operatori Incremento e Decremento come in C.

### Operatori logici come in C.

### Operatori Stringa:

|  |  |
| --- | --- |
| .        Concatenation         $txt1 . $txt2 | Concatenation of $txt1 and $txt2 |
| .=        Concatenation assignment        $txt1 .= $txt2 | Appends $txt2 to $txt1 |

### Operatori Array:

|  |  |
| --- | --- |
| +        Union | Union of $x and $y |
| ==        Equality | Returns true if $x and $y have the same key/value pairs |
| ===        Identity | Returns true if $x and $y have the same key/value pairs in the same order and of the same types |
| !=        Inequality | Returns true if $x is not equal to $y |
| <>        Inequality | Returns true if $x is not equal to $y |
| !==        Non-identity | Returns true if $x is not identical to $y |

### Operatore Ternario come in C.

### If, Else if, Else, Switch come in C.

### For, While, Do While come in C.

### ForEach:

|  |
| --- |
| foreach ($array as $value) {    echo "$value <br>";  } |

### Funzioni:

|  |
| --- |
| function functionName() {  code to be executed;  } |

In PHP 7, type declarations were added. This gives us an option to specify the expected data type when declaring a function, and by adding the strict declaration, it will throw a "Fatal Error" if the data type mismatches.

|  |
| --- |
| <?php  function addNumbers(int $a, int $b) {    return $a + $b;  }  echo addNumbers(5, "5 days");  // since strict is NOT enabled "5 days" is changed to int(5), and it will return 10  ?> |

###### *Valore di ritorno:*

Per definire il valore di ritorno di una funzione si segue la sintassi:

|  |
| --- |
| function addNumbers(float $a, float $b) : float{    return $a + $b;  } |

### Array:

|  |
| --- |
| <?php  $cars = array("Volvo", "BMW", "Toyota");  echo "I like " . $cars[0] . ", " . $cars[1] . " and " . $cars[2] . ".";  ?> |

In PHP, there are three types of arrays:

* **Indexed arrays** - Arrays with a numeric index.
* **Associative arrays** - Arrays with named keys.
* **Multidimensional arrays** - Arrays containing one or more arrays.

The *count()* function is used to return the length (the number of elements) of an array.

###### *Array Associativi:*

Associative arrays are arrays that use named keys that you assign to them.

There are two ways to create an associative array:

|  |
| --- |
| $age = array("Peter"=>"35", "Ben"=>"37", "Joe"=>"43"); |

or:

|  |
| --- |
| $age['Peter'] = "35";  $age['Ben'] = "37";  $age['Joe'] = "43"; |

To loop through and print all the values of an associative array, you could use a foreach loop, like this:

|  |
| --- |
| <?php  $age = array("Peter"=>"35", "Ben"=>"37", "Joe"=>"43");    foreach($age as $x => $x\_value) {    echo "Key=" . $x . ", Value=" . $x\_value;    echo "<br>";  }  ?> |

### Algoritmi Ordinamento:

* *sort()* - sort arrays in ascending order
* *rsort(*) - sort arrays in descending order
* *asort()* - sort associative arrays in ascending order, according to the value
* *ksort()* - sort associative arrays in ascending order, according to the key
* *arsort()* - sort associative arrays in descending order, according to the value
* *krsort()* - sort associative arrays in descending order, according to the key

### Variabili SuperGlobali:

PHP dispone di alcune variabili definite ***SuperGlobali***, ovvero richiamabili da qualsiasi punto e in qualsiasi documento.

The PHP superglobal variables are:

* $GLOBALS
* $\_SERVER
* $\_REQUEST
* $\_POST
* $\_GET
* $\_FILES
* $\_ENV
* $\_COOKIE
* $\_SESSION

###### *$\_SERVER:*

$\_SERVER è un array associativo che contiene varie chiavi, ciascuna chiave definisce una determinata informazioni sul server. Alcune chiavi conute in questo array sono:

$\_SERVER['SERVER\_NAME'];

$\_SERVER['HTTP\_HOST'];

$\_SERVER['HTTP\_REFERER'];

$\_SERVER['HTTP\_USER\_AGENT'];

$\_SERVER['SCRIPT\_NAME'];

…

###### *$\_REQUEST:*

$\_REQUEST is a PHP super global variable which is used to collect data after submitting an HTML form.

The example below shows a form with an input field and a submit button. When a user submits the data by clicking on "Submit", the form data is sent to the file specified in the action attribute of the <form> tag. In this example, we point to this file itself for processing form data. If you wish to use another PHP file to process form data, replace that with the filename of your choice. Then, we can use the super global variable $\_REQUEST to collect the value of the input field:

|  |
| --- |
| <html>  <body>    <form method="post" action="<?php echo $\_SERVER['PHP\_SELF'];?>">    Name: <input type="text" name="fname">    <input type="submit">  </form>    <?php  if ($\_SERVER["REQUEST\_METHOD"] == "POST") {    // collect value of input field    $name = $\_REQUEST['fname'];    if (empty($name)) {      echo "Name is empty";    } else {      echo $name;    }  }  ?>    </body>  </html> |

# PHP Form

## Handling

The PHP superglobals **$\_GET** and **$\_POST** are used to collect form-data.

|  |
| --- |
| <html>  <body>    <form action="welcome.php" method="post">  Name: <input type="text" name="name"><br>  E-mail: <input type="text" name="email"><br>  <input type="submit">  </form>    </body>  </html> |

|  |
| --- |
| <html>  <body>    Welcome <?php echo $\_POST["name"]; ?><br>  Your email address is: <?php echo $\_POST["email"]; ?>    </body>  </html> |

|  |
| --- |
| Output:  Welcome John  Your email address is john.doe@example.com |

### GET vs. POST

Both GET and POST create an array (e.g. array( key1 => value1, key2 => value2, key3 => value3, ...)). This array holds key/value pairs, where keys are the names of the form controls and values are the input data from the user.

Both GET and POST are treated as $\_GET and $\_POST. These are superglobals, which means that they are always accessible, regardless of scope - and you can access them from any function, class or file without having to do anything special.

$\_GET is an array of variables passed to the current script via the URL parameters.

$\_POST is an array of variables passed to the current script via the HTTP POST method.

## Validazione delle Form HTML

Testo alternativo generato dal computer:
PHP Form Validation Example 
* required field 
Name: 
E-mail - 
Website: 
Gender: • Female Male Other * Gender is required 
Submit 
Your Input: 
Gas 
Ciaoo 

**Regole:**

|  |  |
| --- | --- |
| Field | Validation Rules |
| Name | Required. + Must only contain letters and whitespace |
| E-mail | Required. + Must contain a valid email address (with @ and .) |
| Website | Optional. If present, it must contain a valid URL |
| Comment | Optional. Multi-line input field (textarea) |
| Gender | Required. Must select one |

### Validare il nome:

The code below shows a simple way to check if the name field only contains letters and whitespace. If the value of the name field is not valid, then store an error message:

The ***preg\_match()*** function searches a string for pattern, returning true if the pattern exists, and false otherwise.

### Validare l'email:

The easiest and safest way to check whether an email address is well-formed is to use PHP's filter\_var() function.

In the code below, if the e-mail address is not well-formed, then store an error message:

### Validare gli URL:

The code below shows a way to check if a URL address syntax is valid (this regular expression also allows dashes in the URL). If the URL address syntax is not valid, then store an error message:

###### *HTML Form:*

The HTML code of the form looks like this:

|  |
| --- |
| <form method="post" action="<?php echo htmlspecialchars($\_SERVER["PHP\_SELF"]);?>"> |

When the form is submitted, the form data is sent with method="post".

What is the $\_SERVER["PHP\_SELF"] variable?

The $\_SERVER["PHP\_SELF"] is a super global variable that returns the filename of the currently executing script.

So, the $\_SERVER["PHP\_SELF"] sends the submitted form data to the page itself, instead of jumping to a different page. This way, the user will get error messages on the same page as the form.

**In questo modo possiamo mandare i form alla pagina stessa (alla stessa URL) senza conoscere il nome della pagina (codice più manutenibile).**

###### *Sicurezza da Exploit delle HTML Form:*

**The $\_SERVER["PHP\_SELF"] variable can be used by hackers!**

If PHP\_SELF is used in your page then a user can enter a slash (/) and then some Cross Site Scripting (**XSS**) commands to execute.

Cross-site scripting (XSS) is a type of computer security vulnerability typically found in Web applications. XSS enables attackers to inject client-side script into Web pages viewed by other users.

|  |  |  |  |
| --- | --- | --- | --- |
| ESEMPIO:  Assume we have the following form in a page named "test\_form.php":   |  | | --- | | <form method="post" action="<?php echo $\_SERVER["PHP\_SELF"];?>"> |     Now, if a user enters the normal URL in the address bar like "<http://www.example.com/test_form.php>", the above code will be translated to:   |  | | --- | | <form method="post" action="test\_form.php"> |   So far, so good.    However, consider that a user enters the following URL in the address bar:  <http://www.example.com/test_form.php/%22%3E%3Cscript%3Ealert('hacked')%3C/script%3E>    In this case, the above code will be translated to:   |  | | --- | | <form method="post" action="test\_form.php/"><script>alert('hacked')</script> |     This code adds a script tag and an alert command. And when the page loads, the JavaScript code will be executed (the user will see an alert box). This is just a simple and harmless example how the PHP\_SELF variable can be exploited. |

## htmlspecialchars(): Importante

###### *Come evitare $\_SERVER["PHP\_SELF"] Exploits:*

$\_SERVER["PHP\_SELF"] exploits can be avoided by using the ***htmlspecialchars()*** function.

The form code should look like this:

|  |
| --- |
| <form method="post" action="<?php echo htmlspecialchars($\_SERVER["PHP\_SELF"]);?>"> |

# PHP Advanced

### Data and Time:

*date(format,timestamp)*

|  |  |
| --- | --- |
| Parameter | Description |
| format | Required. Specifies the format of the timestamp |
| timestamp | Optional. Specifies a timestamp. Default is the current date and time |

###### *Get a Date:*

The required format parameter of the date() function specifies how to format the date (or time).

Here are some characters that are commonly used for dates:

* d - Represents the day of the month (01 to 31)
* m - Represents a month (01 to 12)
* Y - Represents a year (in four digits)
* l (lowercase 'L') - Represents the day of the week

Other characters, like"/", ".", or "-" can also be inserted between the characters to add additional formatting.

###### *Get a Time:*

Here are some characters that are commonly used for times:

* H - 24-hour format of an hour (00 to 23)
* h - 12-hour format of an hour with leading zeros (01 to 12)
* i - Minutes with leading zeros (00 to 59)
* s - Seconds with leading zeros (00 to 59)
* a - Lowercase Ante meridiem and Post meridiem (am or pm)

### Include:

It is possible to insert the content of one PHP file into another PHP file (before the server executes it), with the include or require statement.

The include and require statements are identical, except upon failure:

* **require** will produce a fatal error (E\_COMPILE\_ERROR) and stop the script
* **include** will only produce a warning (E\_WARNING) and the script will continue

### Gestione File:

The readfile() function reads a file and writes it to the output buffer.

A better method to open files is with the fopen() function. This function gives you more options than the readfile() function.

The **fopen(**) function is also used to create a file

The **fread()** function reads from an open file.

The **fwrite()** function is used to write to a file.

The **fclose()** function is used to close an open file.

### File Upload:

<https://www.w3schools.com/php/php_file_upload.asp>

## PHP Cookies

A cookie is often used to identify a user. A cookie is a small file that the server embeds on the user's computer. Each time the same computer requests a page with a browser, it will send the cookie too. With PHP, you can both create and retrieve cookie values.

### Creare un Cookie:

A cookie is created with the setcookie() function:

|  |
| --- |
| setcookie(*name, value, expire, path, domain, secure, httponly*); |

Only the *name* parameter is required. All other parameters are optional.

The following example creates a cookie named "user" with the value "John Doe". The cookie will expire after 30 days (86400 \* 30). The "/" means that the cookie is available in entire website (otherwise, select the directory you prefer).

We then retrieve the value of the cookie "user" (using the global variable $\_COOKIE). We also use the isset() function to find out if the cookie is set:

|  |
| --- |
| <?php  $cookie\_name = "user";  $cookie\_value = "John Doe";  setcookie($cookie\_name, $cookie\_value, time() + (86400 \* 30), "/"); // 86400 = 1 day  ?>  <html>  <body>    <?php  if(!isset($\_COOKIE[$cookie\_name])) {    echo "Cookie named '" . $cookie\_name . "' is not set!";  } else {    echo "Cookie '" . $cookie\_name . "' is set!<br>";    echo "Value is: " . $\_COOKIE[$cookie\_name];  }  ?>    </body>  </html> |

**Note:** The setcookie() function must appear BEFORE the <html> tag.

### Modificare un Cookie esistente:

To modify a cookie, just set (again) the cookie using the setcookie() function.

### Eliminare un Cookie:

To delete a cookie, use the setcookie() function with an expiration date in the past.

### PHP Session:

A session is a way to store information (in variables) to be used across multiple pages.

Unlike a cookie, the information is not stored on the users computer.

A session is started with the session\_start() function.

Session variables are set with the PHP global variable: $\_SESSION.

|  |
| --- |
| <?php  session\_start();  ?>  <!DOCTYPE html>  <html>  <body>    <?php  // Echo session variables that were set on previous page  echo "Favorite color is " . $\_SESSION["favcolor"] . ".<br>";  echo "Favorite animal is " . $\_SESSION["favanimal"] . ".";  ?>    </body>  </html> |

### PHP Filter:

Validating data = Determine if the data is in proper form.

Sanitizing data = Remove any illegal character from the data.

PHP filters are used to validate and sanitize external input.

The PHP filter extension has many of the functions needed for checking user input, and is designed to make data validation easier and quicker.

The filter\_var() function both validate and sanitize data.

### PHP JSON:

PHP has some built-in functions to handle JSON.

First, we will look at the following two functions:

* json\_encode()
* json\_decode()

# PHP ad Oggetti

### Sintassi:

|  |
| --- |
| <?php  class Fruit {    // Properties    public $name;    public $color;      // Methods    function set\_name($name) {      $this->name = $name;    }    function get\_name() {      return $this->name;    }  }    $apple = new Fruit();  $banana = new Fruit();  $apple->set\_name('Apple');  $banana->set\_name('Banana');    echo $apple->get\_name();  echo "<br>";  echo $banana->get\_name();  ?> |

### This:

The $this keyword refers to the current object, and is only available inside methods.

### instanceof:

You can use the instanceof keyword to check if an object belongs to a specific class:

Ritorna un bool.

|  |
| --- |
| <?php  $apple = new Fruit();  var\_dump($apple instanceof Fruit);  ?> |

### Costruttore:

If you create a \_\_construct() function, PHP will automatically call this function when you create an object from a class.

Notice that the construct function starts with two underscores (\_\_)!

|  |
| --- |
| function \_\_construct($name) {    $this->name = $name;  } |

### Modificatori d'accesso:

There are three access modifiers:

* **public** - the property or method can be accessed from everywhere. This is default
* **protected** - the property or method can be accessed within the class and by classes derived from that class
* **private** - the property or method can ONLY be accessed within the class

### Ereditarietà:

**Ereditarietà multipla non consetita in PHP.**

An inherited class is defined by using the extends keyword.

|  |
| --- |
| <?php  class Fruit {    public $name;    public $color;    public function \_\_construct($name, $color) {      $this->name = $name;      $this->color = $color;    }    public function intro() {      echo "The fruit is {$this->name} and the color is {$this->color}.";    }  }    // Strawberry is inherited from Fruit  class Strawberry extends Fruit {    public function message() {      echo "Am I a fruit or a berry? ";    }  }  $strawberry = new Strawberry("Strawberry", "red");  $strawberry->message();  $strawberry->intro();  ?> |

###### *Parent (Super):*

To call a method from a child class, use the parent keyword inside the child class. Here, the method can be public or protected.

|  |
| --- |
| <?php  class domain {    protected function getWebsiteName() {      return "W3Schools.com";    }  }    class domainW3 extends domain {    public $websiteName;    public function \_\_construct() {      $this->websiteName = parent::getWebsiteName();    }  }    $domainW3 = new domainW3;  echo $domainW3 -> websiteName;  ?> |

### Override dei Metodi Ereditati classico.

### Final nelle classi:

The final keyword can be used to prevent class inheritance or to prevent method overriding.

|  |
| --- |
| <?php  final class Fruit {    // some code  }    // will result in error  class Strawberry extends Fruit {    // some code  }  ?> |

### Costanti nelle Classi:

Constants cannot be changed once it is declared.

Class constants can be useful if you need to define some constant data within a class.

A class constant is declared inside a class with the **const** keyword.

###### *Accedere a costanti da fuori la classe:*

We can access a constant from outside the class by using the class name followed by the scope resolution operator (**::**) followed by the constant name, like here:

|  |
| --- |
| <?php  class Goodbye {    const LEAVING\_MESSAGE = "Thank you for visiting W3Schools.com!";  }  echo Goodbye::LEAVING\_MESSAGE;  ?> |

###### *Accedere a costanti da dentro la classe stessa:*

We can access a constant from inside the class by using the **self** keyword followed by the scope resolution operator (**::**) followed by the constant name, like here:

|  |
| --- |
| <?php  class Goodbye {    const LEAVING\_MESSAGE = "Thank you for visiting W3Schools.com!";    public function byebye() {      echo self::LEAVING\_MESSAGE;    }  }    $goodbye = new Goodbye();  $goodbye->byebye();  ?> |

### Classi Astratte:

Abstract classes and methods are when the parent class has a named method, but need its child class(es) to fill out the tasks.

|  |
| --- |
| <?php  abstract class ParentClass {    public function someMethod1(){  echo "Thank you";  }  **abstract** public function someMethod2($name, $color);  **abstract** public function someMethod3() : string;  }  ?> |

## Tratti: Nuovo concetto presenti nelle classi in PHP

Sono una sorta di **namespace**.

Traits are used to declare methods that can be used in multiple classes. Traits can have methods and abstract methods that can be used in multiple classes, and the methods can have any access modifier (public, private, or protected).

To use a trait in a class, use the '***use***' keyword:

|  |
| --- |
| <?php  trait message1 {  public function msg1() {      echo "OOP is fun! ";    }  }    trait message2 {  public function msg2() {      echo "OOP is fun! ";    }  }    class Welcome {    use message1;  use message2;  }    $obj = new Welcome();  $obj->msg1();  $obj->msg2();  ?> |

### Metodi Statici nelle classi:

Static methods can be called directly - without creating an instance of a class.

Static methods are declared with the static keyword:

|  |
| --- |
| <?php  class greeting {    public static function welcome() {      echo "Hello World!";    }  }  // Call static method  greeting::welcome();  ?> |

A class can have both static and non-static methods. A static method can be accessed from a method in the same class using the **self** keyword and double colon (**::**)

### Parametri (Proprietà) Statici nelle classi:

Static properties can be called directly - without creating an instance of a class.

|  |
| --- |
| <?php  class pi {    public static $value = 3.14159;  }    // Get static property  echo pi::$value;  ?> |

A class can have both static and non-static properties. A static property can be accessed from a method in the same class using the **self** keyword and double colon (**::**)

|  |
| --- |
| <?php  class pi {    public static $value=3.14159;    public function staticValue() {      return self::$value;    }  }    $pi = new pi();  echo $pi->staticValue();  ?> |

To call a static property from a child class, use the ***parent*** keyword inside the child class:

La differenza tra le due sintassi è che:

* La prima conterrà istruzioni php (Es: echo "ciao";)
* La seconda conterrà una espressione, un valore, in php (Es: $variabile)

La prima chiamata sarà una GET quindi questa porzione di codice dentro l'IF verrà saltata.

Ad ogni SUBMIT, quindi tutte le altre chiamate saranno POST (vedi codice Form HTML).

Controlla che la stringa non contenga spazi inutili, \ e caratteri speciali

# Laravel

## Creazione Progetto Laravel

**Laravel** è un framework open source di tipo MVC (Model View-Controller) scritto in PHP per lo sviluppo di applicazioni web.

Per creare un nuovo progetto *Laravel*:

|  |
| --- |
| *$laravel new my\_app* |

Per servire (avviare il server) del progetto:

* Entrare dentro il progetto ***my\_app***

|  |
| --- |
| *$php artisan serve* |

Mostra tutti i comandi:

|  |
| --- |
| *$php artisan* |

## App Laravel semplice, con View

### Struttura Directory:

一 - my-app 
, ~ 500d2 
resources 
public 
Providers 
, Controllers 
, Middleware 
user.php 
Ker ; 

#### Route:

Nella cartella **Route** è presente il file **web.php** che gestirà la navigazione all'iterno della nostra web-app.

*Route::get('/path/...',callback)*

specifica che:

* se dal web arriva alla app una richiesta HTTP GET /path/…  
  ("/path/..." è ciò che, nella URL del browser che genera la richiesta, segue il nome del server).
* la app reagisce invocando callback

Una route definisce quindi la risposta dell'app a una data richiesta HTTP, ossia

instrada ("ruota") la richiesta in arrivo verso il codice che genera la risposta

#### Resources:

Le view (welcome.blade.php) e gli altri "asset" di una app si trovano in resources/

blade è un engine (componente di Laravel) usato per generare pagine web/php, ciascuna delle quali istanzia (e diversifica) una pagina template di base

File di Layout:

Immaginiamo di volere i link "About us" e "Contatteci qui" in ogni pagina/view e non solo nella root (welcome).

Piuttosto che inserirli in ogni pagina, presente e futura, si introduce una vista, qui chiamata layout.blade.php, che farà da "template" per le altre.

Introduciamo la parte parametrica, con l'annotazione @yield (cioè "genera") e diamole il nome, 'contenuto', con cui verrà richiamata, e istanziata in modo specifico, nelle varie pagine basate su layout.

Con @extends('layout') si rende welcome.blade.php un'istanza di layout.blade.php

Introduciamo ora in welcome.blade.php una specifica @section('contenuto') corrispondente all' annotazione @yield('contenuto') in

layout.blade.php

Il contenuto di @section istanzia il corrispondente @yield del template.

###### Doppio @yield:

Nel file di layout si possono introdurre più @yield ...

– p.es. per rendere generico il titolo.

Se una @section, come quella corrispondente allo yield 'titolo', è breve, la si può definire inline, anzichè come blocco.

Per evitare che una view che non istanzia lo @yield('titolo') resti con <title> vuoto, in layout si può anche introdurre un default: @yield('titolo','Laravel').

## App Laravel Avanzata

La view welcome è statica.

Sarebbe utile che la view welcome fosse dinamica, in modo parametrico, cioè che il verbo cambi, da un rendering all'altro, senza che cambi il codice welcome.blade.php, ma solo l'invocazione view('welcome',...) in web.php

Si può, per questo, introdurre un parametro nella view, p.es. $azione\_pref nella view welcome.

– NB: il tag <?=...?> viene rimpiazzato dal valore del PHP.

Ricordando che la view welcome veniva attivata da una route che invoca la funzione view('welcome'), occorre che questa chiamata abbia, per 2° argomento, un array hash (associativo) con chiave 'azione\_pref'.

#### Due parametri nella View:

Vediamo un esempio, con due parametri $azioni\_pref e $titol nella view welcome istanza del template layout.

#### Parametri multipli nella View:

Vediamo un altro esempio, con un terzo parametro $quando nella view welcome. Non lo si può però istanziare dando a view(), nella rotta in web.php, un 4 argomento (causa errore!).

welcome. blade. php -4} 
@extends( layout • ) 
@section( •titolo% $titol) 
@section( •contenuto • ) 
<h2>Cosa ci piace : 
@foreach ($azioni_pref as $una_azione) 
Suna azione; ?> 
@endfo reach 
@endsection 

Bisogna invece dare a view(), come 2° argomento, un array di coppie hash: ognuna di queste ha una:

– chiave nome di un parametro della view (p.es. 'quando') e un

– valore che verrà assegnato al parametro (p.es. 'oggi')

#### Parametri dalla URL:

Il valore da assegnare a una variabile come $quando, nella gestione della route in web.php può anche essere estratto dalla URL, rendendo così l'HTML generato dalla view veramente dinamico.

< ?php 
: I' , function { 
Route 
Sazioni - 
['bere' , 'mangiare' l; 
return view( 'welcone', [ 
azione_pref' $azioni 
quando ' 
request( tempo' 

Per questo scopo si usa la funzione request('id'), che va a cercare un parametro id nella URL.

Nell'esempio qui, request('tempo') estrae dalla URL il parametro ?tempo=oggi

|  |
| --- |
| **{{...}}** non equivale esattamente a **<?= ... ?>**   * Il primo non interpreta il codice JS, è più sicuro. * Il secondo lo interpreta, eseguendolo. |

## Rotte con Chiusure => Controller

Nella *web.php* ogni *callback* che richiama una pagina web statica potrebbe essere raggruppata in un gruppo di *callback*, che chiamaremo **PagesController**.

e! web.php 
e?php 
// keb Routes 
Route: 
Route: 
Route. 
: get( 
: get( 
' / ' , 'PagesControUer@home'); 
' /contact' , 'Pagescontroller@contact'); 

Si genererà quindi una schema da riempire di **controller**, con il tool *artisan*:

|  |
| --- |
| *my\_app $ php artisan make:controller PagesController* |

Una volta creato un Controller navigare alla directory:

|  |
| --- |
| my\_app > app > Http > Controllers > PagesController |

1. Eliminiamo da *web.php* le funzioni di *callback* anonime e sostituiamole con la chiamata al Controller.

|  |
| --- |
| *Route::get('/contact', 'PagesController@contatti');* |

1. Aggiungiamo alla classe **PagesController** tutte le funzioni non-anonime che ci serviranno nel *web.php*

|  |
| --- |
| *public function contatti(){*  *return view('contact', ['titol' => 'Contatti']);*  *}* |

## Modelli

I modelli consentono di accedere in modo uniforme, conciso ed elegante ai BDB che fanno da Back-End all'app Laravel…

L'idea è quella di evitare di scrivere esplicitamente Query SQL ma bensì del codice PHP con oggetti appropriati, che corrispondono a tabelle e record del DB e i cui metodi abbiano effetti "automatici" sul DB stesso.

Tipicamente si sceglie un DB relazione e generalmente si sceglie MySQL.

Tra i file generati alla creazione di un progetto Laravel possiamo notare un file \*.env

che contiene le informazioni riguardo il DB in uso.

Per utilizzare il DB usare phpMyAdmin (installato con xampp)

Ci si porta nella cartella del progetto e si usa il comando *migrate* di *artisan* che creerà una prima tabella, una tabella delle migrazioni, e aggiornerà le tabelle.

|  |
| --- |
| *$ php artisan migrate* |

Bisognerà però impostare il file \*.env e creare su phpMyAdmin un DB con lo stesso riportato nel file.

Questo comando ci da informazione riguardo lo status delle tabelle:

|  |
| --- |
| *$ php artisan migrate:status* |

Questo comando serve per annullare l'ultima migrazione:

|  |
| --- |
| *$ php artisan migrate:rollback* |

#### Modificare campi di una tabella:

Per modificare una tabella bisogna modificare il file del DB presente nel progetto:

|  |
| --- |
| *Database > Migrations > tabella* |

Poi eseguire una ***rollback*** e un ***migrate*** per aggiornare la tabella.

#### Creare una tabella:

Per creare una nuova tabella usiamo il comando

|  |
| --- |
| *$php artisan make:migration <name>* |

Convenzione sul nome: *create\_****<names>****\_table*

### Active Record (pattern):

E' un pattern che fornisce un approccio per l'accesso ai dati di un database tramite codice, in un linguaggio orientato agli oggetti.

Ogni tabella corrisponde, o è gestita come una classe. Un oggetto corrisponde quindi a una riga (o record) della tabella.

L'interfaccia di un oggetto dovrebbe includere:

* Funzioni CRUD: Create, Read, Update, Delete.
* Proprietà corrispondenti (più o meno) alle colonne/attributi di una tabella di DB.

Le relazioni (x:y) tra tabelle e le chiavi delle tabelle sono rispecchiate da apposite proprietà degli oggetti.

L'idea di base è fornire una visione astratta del DB, nascondendo l'interazione con esso.

#### Eloquent:

Gli Active Record (AR) sono implementati attraverso ***Modelli***, creati tramite un componente di Laravel chiamato Eloquent.

|  |
| --- |
| *$ php artisan make:model Project* |

Crea un modello, questo creerà un file Project.php nella directory App del progetto. Il nome del modello sarà maiuscolo, singolare e coinciderà col nome della tabella.

Bisogna quindi creare una tabella con lo stesso nome del **Modello**.

Per popolare una tabella AR si usa un ulteriore strumento di Artisan chiamato **Tinker**.

#### Tinker:

Avviamo la shell tinker, dalla directory del progetto, con:

|  |
| --- |
| *$ php artisan tinker* |

|  |
| --- |
| *>>> App\Project::all()* |

Ritorna tutti i record presenti nella tabella.

Tramite la shell **Tinker** è possibile istanziare e inizializzare oggetti del modello creato con Eloquent…

|  |
| --- |
| *>>> $variabile = new App\Project;* |
| *>>> $variabile->titolo = 'Titolo a caso';* |
| *>>> $variabile->descrizione= 'Descrizione a caso';* |

Per aggiungere o convertire questo oggetto in un Record della tabella di Project, bisognerà lanciare il comando:

|  |
| --- |
| *>>> $project->save();* |

*save()* è un metodo della classe **Model**, creata tramite **Eloquent**.

Interroga il DB:

|  |
| --- |
| *>>> App\Project::latest()->first()->description* |
| *>>> App\Project::all()[0]->title* |
| *>>> App\Project::all()->map->id* |

#### Controller:

Un Controller per il modello…

Per rispondere alle rotte (Route) riguardanti il modello, per esempio '/projects' si introduce un controller nel file *web.php* che gestisce le **Route**.

e! web.php - 
// Routes 
: ' /projects', 'ProjectsControIler@index'); 
Route 
, function 
Route 
return view( 'welcome ) ; 

##### Controller Statici:

Introduciamo nel controller una *index()* che restituisce la **view** *projects.index*, che, per convenzione, sarà il file ***.../resources/views/projects/index.php***

ProjectsController. php 
namespace 
use 
class ProjectsControUer extends Controller 
public function index() { 
return view( 'projects. index' ) ; 

Dobbiamo creare il nuovo controller nella directory:

|  |
| --- |
| *app > http > Controllers > ProjectsController.php* |

##### Controller Dinamici:

###### Stampa di record dal DB:

Si può rendere la risposta dinamica restituendo l'output della query/metodo *all()* applicata al modello **Project** (per ora commentiamo via la pagina view nel file).

c! ProjectsControLIer .php - 
names ce 
use Illurnuna 
p eques , 
class ProjectsControUer extends Controller 
public function index() { 
return Project. : 
return view( 'projects. index'); 

Ma dov'è la classe Project?

Bisogna reperirla nel namespace giusto, cioè: \App!

###### Creazione di record nel DB:

Si introduce ora una **Route** */projects/create*, mappata su un metodo del controller create()

|  |
| --- |
| *<?php*  *Route::get('/projects', 'ProjectsController@index');*  *Route::get('/projects/create', 'ProjectsController@create');*  *Route::get('/', function () { return view('welcome'); });*  *// web.php* |

* Si introduce la view projects.create che corrisponde al file projects/create.blade.php  
  – l'idea è di raggruppare le view relative al modello Project nella dir .../views/projects

< ? ghp 
namespace 
use 
class ProjectsControIIer extends Controller 
public function index() { 
$progetti 
return view( ' projects. index', compact( ' progetti' 
public function create() { 
return view( ' projects. create' 
// ProjectsControIIer .php 

View con form, POST e action:

* viene servita in risposta alla route '/projects/create'
* ciò va bene, ma, quando l'utente, riempito il form, farà clic sul button submit.
* si avrà un errore per mancanza di una route per method e action del form

html» 
eh / head» 
ebodp chL>Crea un 
eform 
" POST" 
type- "text" titLe" 
progetto" 
e/form» 

* Occorre una mappatura per una route /projects e un messaggio HTTP POST :



Store è un nome convenzionale per la funzione **callback** del controller.

namespace 
use 
class ProjectsControtIer extends Controller 
public function index() { 
Sprogetti 
Project 
return view( projects . index' , compact( progetti ' )); 
public function create() 
return view( projects . create ); 
public function store() { 
return request(); 
// ProjectsControILer . pho 

Se provassimo a creare un record adesso otterremo un errore HTTP #419…

La richiesta pervenuta dal cliente con POST è considerata "falsa" se non contiene un token unico, con cui il server individua il client.

Aggiungiamo allora, nella view create, nel form, la funzione PHP csrf\_field()

(Cross-Site Request Forgery, genera token di sicurezza)

La richiesta POST è ora sicura!

c!- create.blade.php - 
c bodys 
cform method; "POST" actionz• [projects •s 
edivscinput typez"text" 
placeholder4TitoIo 

Adesso che la POST è sicura grazie al token, possiamo salvare il record nel DB.

### Gestione dei Controller: Modelli e Risorse

Un Modello è una classe che corrisponde a una tabella del Database.

Per l'utente, un Modello sarà un Risorsa che gli utenti vedono attraverso il browser…

Dobbiamo implementare le funzioni CRUD all'interno del Controller.

Possiamo sostituire tutte le Route di una risorsa con:

|  |
| --- |
| *Route::resource('/project', 'ProjectsController')* |

Tutte le Route verrano gestite dal controller…

Per creare le chiamate di callback del controller posso usare:

|  |
| --- |
| *$php artisan make:controller <name> -r* |

che creerà tutte le funzioni CRUD di callback, le funzioni prenderanno un ID (Ad esempio show).

E' anche possibile creare e associare il Controller con il Model alla creazione del Controller:

|  |
| --- |
| *$php artisan make:controller <name> -r -m <name>* |

che creerà tutte le funzioni CRUD di callback, queste funzioni però prenderanno come parametro un Oggetto e non un ID.

#### PATCH e DELETE:

Non tutti i browser gestiscono appieno questi tipi di messaggi HTTP, che però servono per attivare le route PATCH/DELETE.

Questa riga viene inserita nella fomr per aggirare il browser:

*<input type="hidden" name="\_method" value="DELETE">*

Equivalente alla <input> con hidden può essere utilizzata:

|  |
| --- |
| *{{ method\_field('PATCH') }}* |

oppure

|  |
| --- |
| *@method('DELETE')* |

#### Accesso a record inesistente:

Chiamate a *find($id)*, tipicamente dai callback di un controller, causano errori se non esiste un record con l'*$id* richiesto.

Per evitarli, è meglio usare findOrFail($id), come qui a destra in basso.

Testo alternativo generato dal computer:
public function edit(Sid) // risponde a: HOST/project/I/edtt 
// e mostra form per I (in gener. Std) 
Sproject 
Project: : findOrFaiI(Sid) ; 
return view( 'projects.edit' , cornpact('project')); 

Se fallisce manda il codice HTTP 404|Not Found.

### Modelli, proprietà Fillable e Guarded

Un'altra facility dei modelli è il metodo create(), che si può impiegare nel callback store().

public function store() 
Sproject new Project(); 
Sproject->title request('title'); 
Sproject->description = 
request( ' description ' ) ; 
return redirect( ' [projects' ); 

L'argomento array hash di create() consente di:

* Istanziare il modello, creando l'active record.
* Assegnare in massa tutti gli attributi dell'active record.
* Salvarlo nel database (renderlo persistente).

Ma... i **mass assignment** causano un errore, perché tutti gli attributi che si vogliono assegnare in questo modo devono essere $fillable in \App\<nome>

// lile Project.php 
class Project extends Model 
protected $fillable - 
•title' , 'description ' l; 

La proprietà *$fillable* consente agli attributi specificati di essere riempiti in **Mass Assignment**.

L'alternativa a *$fillable* è dire al modello di non preoccuparsi del *mass assignment* alle proprietà, a meno che siano dichiarate ***$guarded***

// file Project.php 
class Project 
extends Model 
pr otected 
Sguarded — 

Il ***$guarded*** è un array che contiene su quali campi applicare dei controlli di sicurezza.

Tutto ciò serve a proteggere da richieste HTTP malevole, che cercano di assegnare "di nascosto" ad attributi del modello.

Vediamo questo esempio di *store()*:

public function store( 
Project: : 
return redirect( /projects ' 

Essa salva infatti tutti (***all()***) gli attributi inviati con la richiesta.

Se proviamo a iniettare un id nascosto, lo ritroveremo nel database!

## Validazione input

* Rimedio **client-side**: required nei form, ma questo non protegge da manipolazioni, anche maliziose, dell'HTML (chrome developer).

* Rimedio server-side: l'helper request() ha una variante validate():
  + Specificare parametri required nella request HTTP.
  + projects/creote.blode.php • 
    ecocTYPE html. 
    cform method:" POST" 
    type:" 
    Crea 
    c/form-. 
    Serrors Se richiesta non è valida, ridirige automaticamente alla pagina di provenienza, e, per di più, le passa un parametro $errors.

public function store() 
'title' 'required' 
' description' 'required' 

Una comodità importante è ritrovare il testo inserito prima della (eventuale) validazione negativa.

Si usa la helper function **old()**

<form 
type: "text" 
old( •title') 
placeholder:" Titolo progetto" 
etextarea name: "description" 
placeholder:" Descrizione progetto" 
old( 'description') 

**validate()** restituisce ciò che serve, infatti:

public function store() { 
$validated - 
request() - 
title' 
required' , 
max:255'], 
description' 
['required', 'min:8i] 
Project: : create($val idated); 

## Relazioni tra Modelli

A livello di modellazione Entity-Relationship, decidiamo che un Project è fatto di Task.

Usiamo artisan per generare stub del modello e delle tabelle (attraverso le migrazioni).



artisan ci ha dato uno "stub" dell'entità Task aggiungiamo attributi *description*, *completed* e, soprattutto ***project\_id***.

**l'intento è che project\_id stabilisca una relazione con Project.**

Per definire in Laravel la relazione 1 Project → Molti Task, si definisce, dentro la classe Project una funzione tasks():

// file App/Proiect.php 
class Project extends Model 
protected $guarded = [ 
public function tasks() 
return $this->hasmany 
(Task 

Così, dato un Project $proj, si potrà riferire $proj->tasks

#### Cosa fare su phpMyAdmin:

A questo punto su phpMyAdmin dobbiamo creare la relazione tra le due tabelle:

tasks->Struttura->vista relazioni => crea la relazione con il campo **ID** della tabella **projects**.

A questo punto eseguire una **migrate** per apportare le modifiche e modificare la **show()** in *ProjectController* per mostrare i task…

Per i Task occorre:

* una funzione che consenta di simulare un attributo Project che assuma per valore l'unico Project cui il Task appartiene ("belongs to").

class Task extends model 
public function project() { 
return :class) 

In questa maniera possiamo sapere a quale Project appartiene un Task.

## Debugging in Laravel

Possiamo usare il tool Tinker:

|  |
| --- |
| *$ php artisan tinker* |

Tinker lancia una shell psysh che permette di effettuare un debugging a run-time.

|  |
| --- |
| *dd($request->all()); //Funzione di debugging dum-and-die()* |