Struttura HTML

La struttura base dell'HTML è formata da una serie di marcatori che prendono il nome di "tag".
L'insieme dei tag conferisce la tipica struttura ad albero del documento HTML, in cui ogni tag è figlio ad altri fino ad arrivare al tag genitore.
Un tag genitore contiene altri elementi, mentre il figlio quando è contenuto in un altro elemento.

- Il tag `<html>` costituisce la radice del documento HTML, quindi è il contenitore di tutti gli altri tag
- il tag `<head>` contiene informazioni a livello di documento, non necessariamente visibili all'utente, altri tag all'interno di questo si possono trovare
  - `<title>` definisce il titolo della pagina, ci può essere un solo elemento `title`
  - `<meta>` dove si può indicare le parole chiave, autore della pagina, ultima data di modifica e altro.
  - `<script>` dove possono essere incorporati nella pagina oppure collegati tramite file esterni. Utilizzato per JS.
  - `<style>` utilizzato per gestire gli stili di CSS.
- Il tag `<body>` costituisce il corpo del documento, ovvero ciò che il browser visualizzerà.
- Il tag `<main>` specifica il contenuto principale di un documento, ci può essere solo un main nel documento.
- Il tag `<aside>` sono elementi correlati ma "marginali" rispetto al contenuto, utilizzato per sondaggi, link, annunci e altro.
- Il tag `<nav>` è una sezione che permette la navigazione nel sito, tipicamente si mettono delle liste (ma non tutte le liste vanno nel `nav`).
- Il tag `header` utilizzato per materiale introduttivo, usato per header della pagina o header di una sezione o articolo.
- Il tag `footer` contiene le info che vanno alla fine di sezione logica, si può mettere o fondo pagina o fondo articoli.
- Il tag `section` contiene sezioni tematiche, può contenere heading, paragrafi o tutti gli elementi che hanno senso raggruppare.

Cosa si intende per semantica degli elementi HTTP?

L’HTML semantico si riferisce a un codice HTML che utilizza i tag HTML. È un modo efficace per descrivere l’obiettivo dei diversi elementi della pagina Un codice HTML semantico trasmette il significato di ogni elemento e segue gli standard così che chiunque possa capire il codice scritto.

Ad esempio, come `<header>`, `<footer>` e `<article>` sono considerati semantici perché descrivono precisamente lo scopo dell'elemento e il tipo di contenuto al loro interno.

Elementi ancora

`<a href=''> ... </a>`

- Il contenuto dell'elemento diventa attivo
- L'attributo href specifica la URL di un documento nel web(si può specificarlo tramite URL assolute o URL Relative)

Presentazione CSS (Cascading Style Sheets)

È uno standard W3C, definisce la presentazione del documento HTML (ovvero come un documento viene visualizzato in contesti diversi).

```
h1 { color:green; };
```

Dove `h1` è detto *selettore* e identifica l'elemento o gli elementi a cui applicare lo stile.
`Color: green` è detta *dichiarazione* è costituita dalla coppia proprietà valore separati dal `:`

Funziona così:

1. Scrivere documento HTML
2. Scrivere le regole CSS
3. Agganciare le regole all'HTML.
   E il browser visualizzerà lo stile definito

Tipi di selettori

- **Selettore elemento** : seleziono tutti gli elementi di quel tipo, esempio

```
//Seleziono tutti i p

p { color: red;
        text-align: center;
}
```

- **Selettore classe** : seleziono tutti gli elementi di una classe, esempio

```
<h1 class="center"> Heading </h1>
```

```
.center{
        text-align: center;
    color: red;
}

//Oppure

h1 .center {
  text-align: center;
  color: green;

}
```

- **Selettore id** : seleziono l'elemento con quell'id, esempio

```
<p id="para1"> ... </p>


#para1 {
        text-align: center;
    color: red;
}
```

Relazioni nel DOM

- **Discendenti**
  - gli elementi contenuti in un elemento sono i suoi discendenti
- **Figli**
  - discendenti diretti e viceversa si dice genitore
- **Antenati**
  - gli elementi sopra nell'albero
- **Genitori**
  - elementi direttamente sopra
- **Fratelli**
  - elementi con lo stesso genitore

Discendenza

Come sappiamo gli elementi di una pagina costituiscono una struttura ad albero. I selettori permettono di applicare lo stile in base a quest’ultimo:

Ad esempio mediante comando css `main p{color: grey;}` andremo a modificare lo stile di tutti i discendenti di main del tipo `<p>` (Quindi tutti i paragrafi presenti nel main e nei sotto-elementi del main dovranno rispettare tali regole).

I selettori introdotti si possono poi combinare, ad esempio con `class1 p{..}` andiamo a applicare le regole a tutti i paragrafi presenti nei discendenti della classe `class1`.

Ad esempio, selettore di discendenti: `#contenitore p {color: white;};` va letto da destra a sinistra, il codice assegna ai paragrafi `p` contenuti nel div `#contenitore`, ovvero ai paragrafi discendenti del div con id contenitore.

```
<div id="contenitore">

<p>...</p>

</div>

<div id="box">

<p>...</p>

</div>
```

Qui solo il primo p rispetta questa condizione.

Altro esempio, selettore di figli: padre > figlio {dichiarazioni;} ; il selettore > consente di selezionare un elemento che è figlio diretto dell'elemento padre.

```
<div id="box">

<p>Primo paragrafo</p>

<div>

<p>Secondo paragrafo</p>

</div>

<p>Terzo paragrafo</p>

</div>
```

Qui solo il primo e terzo sono figli diretti del div con id `#box`. Se facessi `#box > p {color: white}` solo loro avranno il testo bianco.

Altro esempio, selettore fratelli adiacenti: fratello + fratello adiacente {dichiarazioni;} ; Permette di assegnare regole CSS agli elementi che si trovano sullo stesso livello del DOM.

```
<div>

<h1>1. Titolo principale h1.</h1>

<h2>1.1 Primo sottotitolo h2.</h2>

<p>...</p>

<h2>1.2 Secondo sottotitolo h2.</h2>

<p>...</p>

</div>
```

Qui applicando, h1 + h2 {color: white;} , verrà selezionato solo il primo h2 dato che è immediatamente adiacente al tag h1.
MA può succedere che dato

```
<ul>

<li>List Item 1</li>

<li>List Item 2</li>

<li>List Item 3</li>

<li>List Item 4</li>

<li>List Item 5</li>

</ul>
```

E dato li + li {color: white;} , solo il primo li non coinciderà con la proprietà.

Esempio, selettore generale di fratelli : fratello ~ fratello {dichiarazioni;} ; assegna uno stile a tutti gli elementi che sono fratelli. Dato:

```
<div>

<h1>1. Titolo principale h1</h1>

<h2>1.1 Primo sottotitolo h2</h2>

<p>...</p>

<h2>1.2 Secondo sottotitolo h2</h2>
```

```
<p>...</p>

</div>
```

E dato, h1 ~ h2 {color: white;}, seleziona tutti gli elementi h2 dello stesso livello di h1 indipendentemente dalla posizione che occupano.
Invece dato,

```
<div>

...

<h2>Sottotitolo h2</h2>

<h3>Sottotitolo h3</h3>

...

<h2>Sottotitolo h2</h2>

<h3>Sottotitolo h3</h3>

...

<h2>Sottotitolo h2</h2>

<h3>Sottotitolo h3</h3>

</div>
```

E dato, h3 ~ h2 {color: white;}, al primo h2 non viene applicato nulla mentre agli altri si, questo è dato dal fatto che la regola viene assegnata solo agli elementi che sono fratelli e successori dell'elemento h3.

Ereditarietà

Le regole applicate agli elementi si riflettono anche sui sotto-elementi che questi contengono.
ES: se si applica un particolare elemento di stile al body, questo verrà ereditato da ogni elemento nel corpo della pagina.

Cascade

I browser hanno stili predefiniti che entrano in conflitto con le regole introdotte dall'utente. Per risolvere questo problema sono stati introdotti appositi algoritmi che selezionano le proprietà che poi verranno effettivamente applicate. I conflitti sono molto comuni e possono essere di vario tipo:

- Diretti: quando ad esempio si hanno due proprietà nella stessa regola che modificano entrambe lo stesso tratto (ad es. il colore): in questo caso l'algoritmo selezionerà l'ultima proprietà definita.
  Ciò accade anche quando le proprietà si trovano in regole diverse: verrà comunque selezionata l'ultima.
- Di tipo: le regole definite per classi hanno priorità rispetto a quelle definite per elementi. Inoltre si **ha che le priorità definite sulla linea mediante `<style>` saranno prioritarie su tutte le altre.** Allo stesso modo ID > class (Mediante '!important' è possibile impostare proprietà che saranno applicate a prescindere dalle altre).

La specificità del css?

La specificità è una serie numerica che permette ai browser di capire quale regola CSS ha la priorità sulle altre che cercano di selezionare lo stesso elemento. Prende il controllo la proprietà più vicina all'oggetto.

Ad ogni dichiarazione è attribuita una specificità misurata con quattro valori `[a,b,c,d]`. Dove "a" è il valore più importante e "d" il meno importante. Si confrontano i valori più importanti e se uguali si passa a quelli successivi altrimenti termina.

Per specificità si intende la priorità che una regola ha sulle altre che si riferiscono allo stesso selettore. La proprietà con valore di specificità maggiore avrà la precedenza sulle altre; nel caso in cui due o più regole, come in precedenza, abbiamo tale valore uguale, verrà assegnata l'ultima inserita nell'ordine a cascata.

- Calcolo dei valori
  a. 1 se la dichiarazione è inline, 0 altrimenti
  b. numero di selettori id
  c. numero di selettori di classe, attributo o pseudo-classe
  d. numero di selettori elemento o pseudo elemento

Font-family

Una famiglia di caratteri è un insieme di caratteri che hanno un design comune. I caratteri all'interno di una famiglia, tuttavia, differiscono l'uno dall'altro nello stile come il peso (leggero, normale, grassetto, semi-grassetto, ecc.).

Tutti i font cominciano con la lettera maiuscola tranne quelli generici.

Se il nome contiene uno spazio allora va messo tra le virgolette. Sono separati da virgole.

Cosa è il font stack?

Praticamente quando noi impostiamo diversi font se il primo non dovesse essere installato verrà utilizzato il secondo e così via, creando appunto una sorta di stack.
Esempio: `font-family: Verdana, Geneva, sans-serif;`

Cosa sono i web font?

Sono caratteri tipografici che vengono caricati da un server web e utilizzati per visualizzare il testo su una pagina web.

Quindi sono caratteri esterni a quelli di default.

Vengono importati con la regola `@font-face { font-family: font }`

Per introdurre font personalizzati come occorre procedere? Ci sono 2 strategie:

-**Font ospitati localmente:** i file dei font vengono caricati sul server web e poi utilizzati nella pagina .css mediante il comando @fontface. Il browser quando deve caricare la pagina inoltrerà al server richieste http per ricavare i font specificati nel .css ed a questo punto il SO li renderizzerà per permettere il caricamento e la visualizzazione della pagina

-**Font ospitati esternamente**: i file dei font in questo caso sono memorizzati su server remoti. In tal caso la pagina web non dovrà archiviare al proprio interno i file dei font, ma solo un collegamento a questi ultimi (NOTA: I collegamenti sono solitamente forniti da servizi di hosting dei server remoti).

Font-size

Specifica la dimensione del carattere

Gestione del Box

Box model

- Ogni elemento html è contenuto in un box rettangolare

Differenza tra margin e padding

Margin è una proprietà CSS che definisce uno spazio tra un box e ciò che lo circonda all'interno della pagina web.

Padding indica, lo spazio tra il contenuto e il box che lo contiene, all'interno di eventuali bordi definiti.

Border-box

Il valore border-box (al contrario del valore di default content-box) rende il bordo l'elemento ultimo del nostro box e fa sì che la larghezza consideri padding e bordi come interni.

Cosa è il collasso dei margini?

Quando due o più margini verticali (cioè margini superiori o inferiori) di elementi blocco si incontrano, si verifica il collasso. Invece di sommare i due margini, il browser applica solo il margine più grande.

delle soluzioni per evitare il collasso possono essere: sono elemento float o absolute.

Posizione degli elementi

CSS Position

- **Static** : L'elemento è posizionato in base al normale flusso del documento, e non è influenzato dalle proprietà `top` , `bottom` , `right` e `left` .
- **Relative** : L'elemento è posizionato rispetto alla sua posizione normale. Impostando le proprietà `top` , `bottom` , `right` e `left` verrà regolato in modo da allontanarsi dalla sua posizione normale.
- **Fixed** : L'elemento è posizionato relativamente alla viewport, ovvero rimane sempre allo stesso posto anche se si scorre la pagina. Le proprietà `top` , `bottom` , `right` e `left` sono utilizzate per posizionare l'elemento.
- **Absolute** : L'elemento è posizionato rispetto all'antenato più vicino. Se non dovesse avere un antenato, utilizza il corpo del documento e si sposta insieme allo scorrimento della pagina.

Float

La proprietà CSS posiziona un elemento sul lato sinistro o destro del suo contenitore, consentendo al testo e agli elementi in linea di avvolgerlo. L'elemento viene rimosso dal normale flusso della pagina, pur rimanendo parte del flusso (in contrasto con il posizionamento assoluto).

Oppure

Questa proprietà viene utilizzata per posizionare e formattare il contenuto, ad esempio per far si che un immagine fluttui a sinistra del testo in un contenitore.

- Si staccano dal flusso normale ma influenzano il contenuto dei blocchi intorno.
- Sono contenuti nell'area del contenuto dell'elemento che li contiene.
- I margini sono mantenuti.

FlexBox Display

Flexbox

Diciamo che permette di avere una "scatola flessibile" . Il modulo Flexbox Layout semplifica la progettazione di una struttura di layout flessibile e reattiva senza utilizzare float o posizionamento.

Media query

Definiscono degli stili per determinati media e device.
Utilizzati nei siti responsive fornendo layout di versi a seconda della dimensione della finestra di visualizzazione.

Desktop first e mobile first?

I dispositivi mobile possono presentare caratteristiche dello schermo, come risoluzione e dimensione, molto diverse fra loro. Questo implica che nella costruzione della pagina web un’intera sezione della fase di progettazione dovrà essere dedicata all’implementazione di regole che rendano il contenuto ‘flessibile’ e quindi visualizzabili nei vari schermi a disposizione.

L’obiettivo è quello di fornire layout diversi a seconda della dimensione della finestra di visualizzazione.

L’approccio basato su Responsive Web Design si occupa proprio di questo. Tecniche utilizzate:

-controllo viewport

-controllo layout con media queries

-stili fluidi → utilizzo di % come unità di misura per definire la grandezza degli elementi (anche img o video)

NOTA → **Strategia mobile-first:** definire prima le regole css per i dispositivi mobili e poi tramite @media(min-width: -) impostare regole per i desktop.

Tecniche responsive

Le tecniche utilizzate sono:

- **Controllo del Vlewport** : bisogna sempre inserire nella `head` il meta tag `viewport` , suggerendo così al browser come dovrà gestire la viewport.

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

  Opzioni :
  – width, height -> device-width o device-height oppure px
  – user-scalable -> no, yes
  – initial-scale -> 1
  – minimum-scale -> 1 non scala
  – maximum-scale -> 1 non scala
- **Controllo Layout con media query** : Vengono utilizzati i breakpoints, sono dei punti su una scala ideale di larghezza del viewport in cui si verifica una qualche modifica al layout della pagina. Essi si definiscono con valori numerici nella media query con `max-width` e `min-width`.
- Siti fluidi : utilizzo la `%` come unità di misura per definire la grandezza degli elementi (anche img o video).

Caratteristiche di Javascript?

Javascript è un linguaggio di programmazione sviluppato per rendere interattive le pagine web. Si tratta di un linguaggio non compilato che gira su VirtualMachine.

In generale può tornare utile per:

-aggiungere classi css in base a eventi generati dall'utente

-comunicazione asincrona → invio dati senza ricaricare la pagina

Quindi cosa fa:

- Modificare elementi della pagina
- Interagire con un server remoto
- Reagisce ad azione dell'utente
- Imposta cookie e contenuti locali
- Disegna sulla pagina

Cosa non fa:

- Accede ai file locali del computer
- Interagisce con qualunque server remoto

Le caratteristiche sono :

- Dynamic: non è compilato, gira in una macchina virtuale
- Loosely typed: non bisogna dire che tipo ha una variabile
- Case-sensitive

Comportamento (JS)

Transpilers

Vengono utilizzati per tradurre linguaggi differenti in un a specifica versione di JS (source-to-source translator).
Quindi permettono la scrittura dei programmi in diversi modi, più comodi per alcuni utenti.

Polyfill

Sono librerie JS che adattano del codice (html, css, js) per renderlo retrocompatibile con standard "vecchi".
Ad esempio: HTML5, rende una pagina HTML5 compatibile con vecchi browser non HTML5.

Garbage collector

Un Garbage Collector rimuove le variabili che non ci servono dalla memoria automaticamente.
Capisce quando non abbiamo più riferimenti ad un oggetto, possiamo dichiarare nuove variabili dinamicamente senza preoccuparci di rimuoverle dalla memoria.

Quindi, tutte le variabili che creiamo occupano memoria che viene allocata dinamicamente, un algoritmo (garbage collector) capisce gli oggetti non più raggiungibili dallo script e rilascia la memoria

Cos'è un metodo

Praticamente le funzioni memorizzate nelle proprietà degli oggetti sono detti *metodi*.
Permettono agli oggetti di "fare" qualcosa come `object.sayHi()`.
Esempio:

```
let v = "ciao"  //Dichiarazione

v.toUpperCase();  //Metodo
```

Per quanto riguarda GET e POST invece i metodi http specificano la tipologia della richiesta.

Strict mode

Lo scopo `"use strict"` è indicare che il codice deve essere eseguito in "modalità rigorosa". Con la modalità rigorosa non è possibile, ad esempio, utilizzare variabili non dichiarate.

Funzioni

Un modo per raggruppare dei comandi e per richiamare più volte lo stesso codice.

Scope

Lo scope è la visibilità di una variabile (la regione di codice dove possiamo usare il nome della variabile/funzione)

- Variabili locali definite dentro una funzione hanno lo scope relative al blocco della funzione stessa (detta *local scope*)
- Quando definiamo una variabile fuori da ogni funzione, è detta *global scope* e diventa visibile ad ogni altro js che gira nella pagina.

```
// global scope
let a = 5;
function x() {

return a + 1; }

x();// ritorna 6
a; // 5
//Fine global


// local scope
function x() {

//Local
let a = 5;

return a + 1; }
```

```
//fine local

x();// ritorna 6
a; // undefined
```

Parametri

I parametri mancanti sono impostati a *undefined*
Esempio:

```
function somma(a, b) {
  let somma = a + b;
  return somma;

}
//b = undefined -> 3 + undefined = NaN
let s = somma(3);
```

Arrow Functions

È un sistema più sintetico di specificare una funzione

```
let somma = (a,b) => a + b;
```

Oggetti

Un oggetto è una lista di coppie “proprietà” “valore”, racchiuse in parentesi angolari { }

Come faccio a creare oggetti?

Un oggetto può essere creato con parentesi `{}` con un elenco facoltativo di proprietà. Una proprietà è una coppia "chiave: valore", dove la chiave è una stringa e valore può essere qualsiasi cosa.
Ad esempio:

```
let user = {
        name: "John",
        age:30
};
```

Metodi

Un oggetto può avere tra le sue proprietà anche delle funzioni che chiameremo metodi.
Esempio:

```
let a = {"name": "pippo"};

a.saluta = function () {
    alert("Ciao sono pippo");

};

a.saluta()
```

This

Può essere utile nei metodi riferirci ad altre proprietà dell'oggetto.
Esempio:

```
let a = {"name": "pippo"};

a.saluta = function () {
    alert("Ciao sono " + this.name);

};
```

This lato function

La parola *this* si riferisce al contesto in cui un pezzo di codice, come il corpo di una funzione, deve essere eseguito. Di solito utilizzato nei metodi degli oggetti, dove this si riferisce all'oggetto a cui il metodo è collegato, consentendo così di riutilizzare lo stesso metodo su oggetti diversi.

Costruttori

Per creare oggetti uguali o simili possiamo usare delle funzioni, quando viene chiamato un costruttore con *new*:

- viene creato un oggetto vuoto e assegnato a *this*
- viene eseguita la funzione
- viene ritornato this

Array

- Contenitori di variabili, anche con tipi diversi
- Sono oggetti con proprietà numeriche e metodi/attributi per “maneggiarli”
- Ogni elemento dell’array ha un indice (che parte da 0)

```
var myFirstArray = [5, “ciao”, false, undefined];
```

Creare un array (metodi equivalenti)
– `let arr = [element0, element1, ..., elementN];`

- Modificare un membro
  – `myFirstArray[0] = “nuovo valore”;`
- Aggiungere un membro
  – `myFirstArray.push(“ciao”) //aggiunge alla fine`
  – `myFirstArray.unshift(“ciao”) //aggiunge all'inizio`
  – `myFirstArray[10] = “ciao”; // aggiunge al decimo posto (length sarà almeno 11)`
- Rimuovere un membro
  – `myFirstArray.pop() // rimuove ultimo elemento ritornandolo`
  – `myFirstArray.shift() // rimuove il primo elemento ritornando`
  – `delete myFirstArray[10] // rimuove l’elemento ma non sposta gli indici dell’array`
- Lunghezza dell’array
  – `myFirstArray.length`
- Svuotare un array
  – `myFirstArray = []`
  – `myFirstArray.length = 0`

Window

L'oggetto window rappresenta la finestra del browser.
Oltre a alert, confirm, prompt, l’oggetto window ha altre funzioni (“metodi”) utili:

• setTimeout(funzione_da_chiamare, time): richiama la funzione scelta dopo time millisecondi

• setInterval(funzione_da_chiamare, time): richiama la funzione ciclicamente dopo time ms

• clearTimeout e clearInterval interrompono le funzioni precedenti

Eccezioni

All’interno di una funzione, in caso di valori indesiderati oppure altri errori generici, è possibile lanciare una eccezione mediante il comando *throw.*

Se la funzione viene poi chiamata, l’eccezione può essere catturata mediante il blocco try{funzione()} e catch(e) {}.

Var e Let

La differenza tra le due modalità di dichiarazione è sostanzialmente una→ lo scope di let è il blocco di operazioni in cui è definita (for / if-else / …) mentre lo scope di Var è l’intera funzione in cui si trova. Se una variabile viene dichiarata senza queste due keyword, diventa proprietà dell’oggetto predefinito window (può causare errori).

Closure

Il principio di base su cui si fonda questo meccanismo stabilisce che ogni variabile che era accessibile quando una funzione è stata definita rimane *"racchiusa"* nello scope accessibile dalla funzione. Questo meccanismo è detto **closure**.

Lo scope viene creato quando una funzione viene chiamata, mentre la closure viene creata durante la definizione di una funzione. In questo modo quando una funzione viene chiamata, la closure già esistente garantirà l'accesso agli scope disponibili nelle altre funzioni annidate in base al livello di annidamento.

```
function salutatore(name) {
    let text = 'Ciao' + name; // Local variable
    let diCiao = function() { alert(text); }
    return diCiao;

}
let s = salutatore('Lorenzo');
s(); // alerts "Ciao Lorenzo"

"s" non memorizza solo il return della funzione "salutatore" (che è una
funzione), ma anche le variabili appartenenti al suo scope (ad es. la
variabile "text")
```

Prototipo

Ciascuna classe presenta al suo interno una proprietà, detta appunto *Prototipo*, qualsiasi oggetto che fa parte di questa classe sarà istanziato esattamente come una copia di questo prototipo.

Modificando questa meta classe sarà possibile influenzare direttamente tutti i precedenti oggetti creati a partire da questa classe.

Prototipo di oggetti

È una sorta di riferimento ad un altro oggetto. Gli oggetti che creiamo hanno come prototipo `Object`, mentre quando creiamo un oggetto tramite un costruttore, il suo prototipo è l'oggetto `prototype` del costruttore.
Quindi è l'istanza dell'oggetto dal quale l'oggetto è ereditato.

Come associare un prototipo ad un oggetto?

```
function Student(name, age) {
  this.name = name;
  this.age = age;

}
Student.prototype.university = "Tor Vergata";

let pippo = new Student("Pippo", 20);
```

Cosa succede?

1. Viene creato un nuovo oggetto vuoto
2. Viene passato al costruttore (function Student), in modo che ci possa riferire con “this”
3. Il costruttore setta le proprietà dell’oggetto
4. Il costruttore imposta: prototipo dell’oggetto creato = prototipo della funzione Student.prototype à pippo.**proto**

Prototipo di funzione

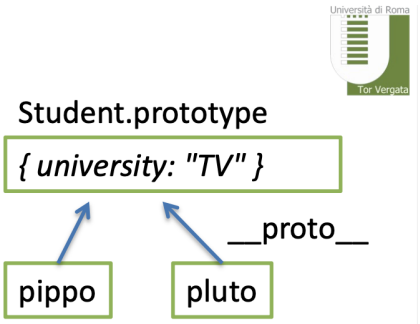
È l'istanza di un oggetto che diventerà il prototipo per tutti gli oggetti creati usando il metodo costruttore

Ogni funzione ha la proprietà `protype` il cui valore è un oggetto.
Scrivendo `Student.prototype.university = "Tor Vergata`, viene assegnata una proprietà a quell'oggetto.
Tutti gli oggetti creati con questo costruttore, avranno come prototipo il prototipo della funzione

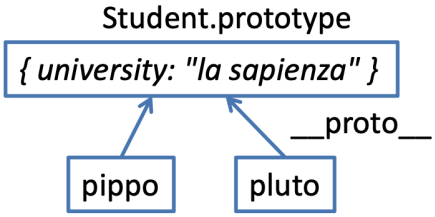
Quindi per i prototipi

- Prototipi di funzione: È l'istanza di un oggetto che diventerà il prototipo per tutti gli oggetti creati usando la funzione come costruttore, ovvero `NomeFunzione.prototype`
- Prototipi di oggetto: È l'istanza dell'oggetto dal quale l'oggetto è ereditato `NomeOggetto.__proto__`

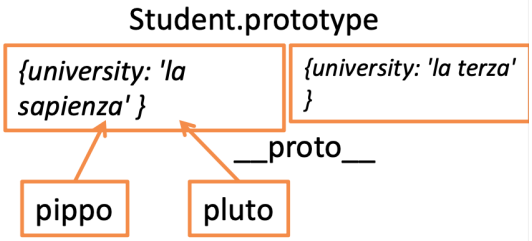
```
function Student(name, age) {
  this.name = name;
  this.age = age;
}
Student.prototype.university = "TV";

let pippo = new Student("Pippo", 20);
let pluto = new Student("Pluto", 21);

pippo.university;
pluto. university;
```




```
Student.prototype.university =
"La Sapienza";
pippo.color;
pluto.color;
```




```
Student.prototype = {university:
"La terza"};
pippo.color;
pluto.color;
```



DOM (Document Object Model)

È un interfaccia di programmazione per HTML.
Fornisce una mappa strutturata del nostro documento ed i metodi per interfacciarsi con gli elementi.

- Ogni elemento della pagina è un nodo
- l'elemento radice è "document"
- document ha una serie di proprietà standard.

Ad esempio si può selezionare un elemento:

- `document.getElementById("miodiv")`, ritorna il node associato al div con id "miodiv"
- `document.getElementByTagName("p")`, ritorna un NodeList degli elementi "p"
- `document.getElementByClassName("myclass")`, ritorna elementi con classe "myclass"

- `document.querySelectorAll("p .warning")`, permette di usare selettori css e ritorna una NodeList(una NodeList è simile a un'array)

Oppure associare un evento a un elemento:

1. con un attributo HTML `<body onclick="myFunction();”>`
2. con un metodo,
   `window.onclick = myFunction;`
   `document.getElementById(”miodiv”).onclick = ...`
3. con addEventListener, `window.addEventListener("click", myFunction);`

Come si manipola un nodo? Dopo aver selezionato un elemento *myElement* mediante i comandi che già conosciamo, possiamo procedere nel seguente modo:

-*myElement.setAttribute*(‘src’, ‘image.img’); → possiamo impostare una immagine in un tag

-*myElement._style.backgroudColor* = “#fff”; → modifichiamo lo stile

-*let myElement* = document.createElement(“div”); → creiamo un nuovo elemento. Per aggiungerlo ad una certa parte della pagina, otteniamo *oldElement* mediante selettori e poi usiamo *oldElement._appendChild(myElement)*;

-*oldElement.insertBefore*(*myElement, para*) → inseriamo nuovo elemento prima di para

-*oldElement._replaceChild(newImg, oldImg)* → *sostituzione elemento* - *_parent._removeMe(_child*) → rimuoviamo dal genitore l’elemento figlio specificato.

Codice sincrono e asincrono

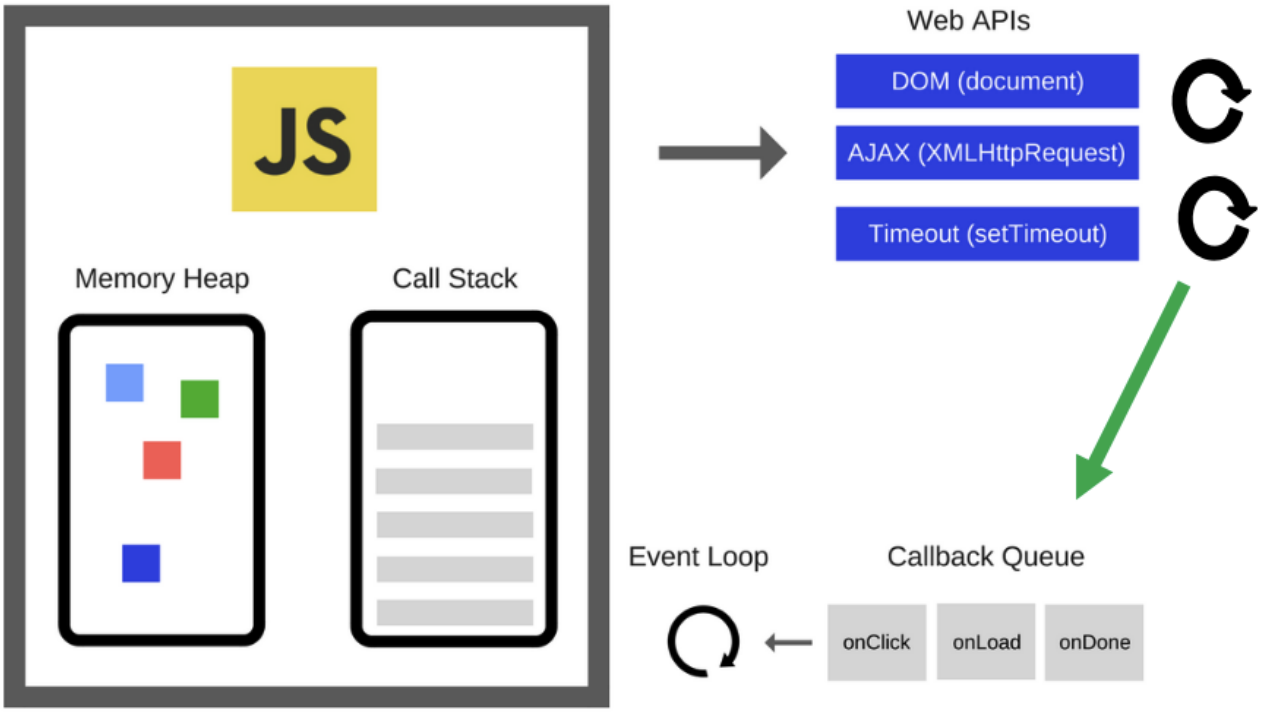
Il codice sincrono è eseguito linea dopo linea :

- Ogni linea aspetta che finisce l'esecuzione della precedente
- Le operazioni lunghe bloccano l'esecuzione del programma

Il codice asincrono è eseguita alla fine dell'esecuzione del task:

- Il codice sincrono continua la sua esecuzione
- Le immagini sono caricate in modo asincrono

Event Loop



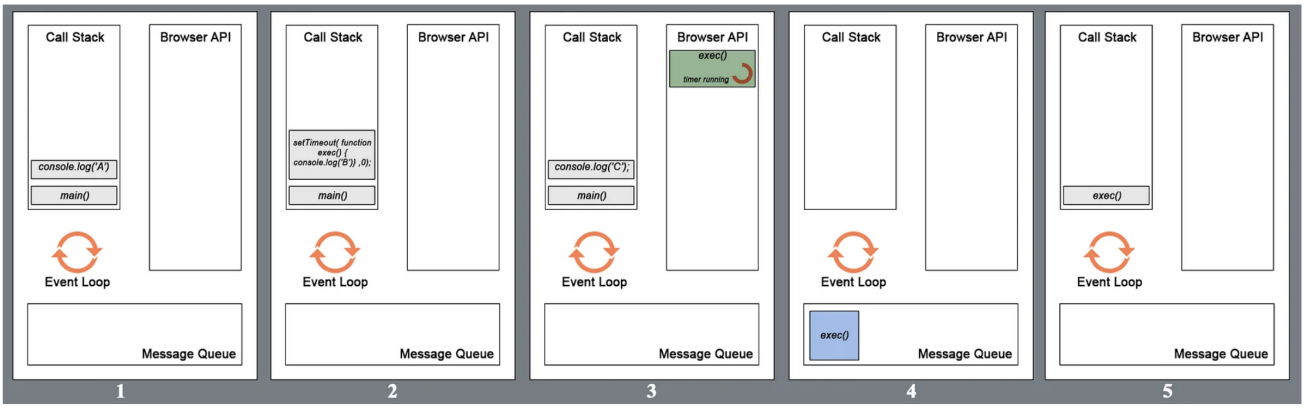
Il motore è costituito da due componenti principali:

- Heap di memoria: è qui che avviene l'allocazione della memoria
- Stack di chiamate: è qui che si trovano i frame dello stack durante l'esecuzione del codice,Il Call Stack è una struttura dati che registra fondamentalmente in quale punto del programma ci troviamo.
- API nel browser: sono fornite dai browser, come DOM, AJAX e altro. E poi c'è il ciclo di eventi e la coda di callback.

Ad esempio, dato questo codice:

```
function main(){
console.log('A');
setTimeout(
function display(){ console.log('B'); }
,0);
console.log('C');
}
main();
```

Qui abbiamo la funzione principale che ha 2 comandi console.log che registrano 'A' e 'C' sulla console. Tra di essi c'è una chiamata setTimeout che registra 'B' sulla console con un tempo di attesa di 0 ms.



1. La chiamata alla funzione principale viene prima inserita nello stack (come frame). Quindi il browser inserisce la prima istruzione nella funzione principale nello stack, ovvero console.log('A'). Questa istruzione viene eseguita e al termine viene estratto quel frame. L'alfabeto A viene visualizzato nella console.
2. La successiva istruzione (setTimeout() con callback exec() e tempo di attesa 0 ms) viene inserita nello stack delle chiamate e l'esecuzione inizia. La funzione setTimeout utilizza un'API del browser per ritardare una callback alla funzione fornita. Il frame (con setTimeout) viene quindi estratto una volta completato il passaggio al browser (per il timer).
3. console.log('C') viene inserito nello stack mentre il timer è in esecuzione nel browser per il callback alla funzione exec(). In questo caso particolare, poiché il ritardo fornito era di 0 ms, il callback verrà aggiunto alla coda dei messaggi non appena il browser lo riceverà (idealmente).
4. Dopo l'esecuzione dell'ultima istruzione nella funzione principale, il frame main() viene estratto dallo stack delle chiamate, rendendolo vuoto. Affinché il browser possa spingere un messaggio dalla coda allo stack delle chiamate, lo stack delle chiamate deve prima essere vuoto. Ecco perché, anche se il ritardo fornito in setTimeout() era di 0 secondi, il callback a exec() deve attendere che l'esecuzione di tutti i frame nello stack delle chiamate sia completa.
5. Ora il callback exec() viene inserito nello stack delle chiamate ed eseguito. L'alfabeto C viene visualizzato sulla console. Questo è il ciclo di eventi di javascript.

AJAX (Asynchronous JavaScript And XML)

AJAX usa solo una combinazione di:

- Un oggetto XMLHttpRequest incorporato nel browser (per richiedere dati da un server Web)
- JavaScript e HTML DOM (per visualizzare o utilizzare i dati) –
- AJAX è il sogno di uno sviluppatore, perché può:
  - Aggiornare una pagina Web senza ricaricare la pagina
  - Richiedere dati a un server - dopo che la pagina è stata caricata

- Ricevere dati da un server - dopo che la pagina è stata caricata
- Inviare dati a un server - in background

XMLHttpRequest

- Crea una richiesta web
- Metodi/Attributi più utilizzati:
  - open('GET', 'http://www.uniroma2.it', false). Il terzo parametro dice se la richiesta deve essere asincrona (ovvero se la funzione si deve bloccare finché non ritorna). Se async=true, dobbiamo gestire manualmente utilizzando listener di eventi
  - send(), Invia la richiesta
  - responseText , la risposta (DOMString)

Fetch

- Sono API basate sulle promise per le richieste AJAX
- sostituiscono le XMLHttpRequest, poichè fetch() ha una sintassi più semplice
- supportato da tutti i browser moderni

L'API prevede una gestione delle chiamate asincrone basata sulle *promise*

Cos'è una Promise

Una promise è un oggetto usato come placeholder per il risultato futuro di un operazione asincrona.
SI può vedere come un contenitore usato per il risultato futuro che può essere positivo o negativo.

Permette anche di collegare una *producing code* con una *consuming code*

- Si può creare così `let promise = new promise( function (resolve,reject )) {}`

Come creare una promise

```
let promise = new Promise(function(resolve, reject){
        //la funzione è eseguita automaticamente quando la promise è costruita

        //Dopo 1 secondo il segnale che ha lavorato darò il risutalto "done"
```

```
        setTimeout(() => resolve("done"), 1000);
});
```

Consumare una promise: then

```
let promise = new Promise(function(resolve, reject){
        setTimeout(() => resolve("done"), 1000);
});

promise.then(
        result => alert(result),
        error => alert(error)
);
```

Microtask

Per prima cosa, ogni volta che un task esce, il ciclo degli eventi controlla se il task sta restituendo il controllo ad altro codice JavaScript.In caso contrario, esegue tutti i microtask nella coda dei microtask. La coda dei microtask viene, quindi, processata più volte per ogni iterazione del ciclo degli eventi, anche dopo aver gestito eventi e altri callback.

In secondo luogo, se un microtask aggiunge più microtask alla coda chiamando queueMicrotask() , quei microtask appena aggiunti vengono eseguiti prima che venga eseguita l'attività successiva . Questo perché il ciclo degli eventi continuerà a chiamare i microtask finché non ne rimangono più in coda, anche se ne vengono aggiunti altri.

La ragione principale per usare i microtasks è questa: assicurare un ordine coerente dei compiti, anche quando i risultati o i dati sono disponibili in modo sincrono.

Coda Microtask

Le attività asincrone necessitano di una gestione adeguata. La coda microtask ha questa specifica:

- È first-in-first-out, quindi le attività che entrano prima vengono eseguite per prime.
- L'esecuzione di un'attività avviene solo quando non è in esecuzione nient'altro. vabb
- La coda dei Microtask ha priorità su quella delle callback.

Si può dire anche così, quando una promessa è pronta i suoi gestori `.then/catch/finally` vengono messi in cosa, non vengono ancora eseguiti. Quando il motore JS si libera dal codice corrente, prende un'attività dalla coda e la esegue.

Cos'è CORS?

CORS sta per **Cross Origin HTTP Request** e una web application con dominio X non può richiedere dati ad un dominio Y tramite AJAX se non ha abilitato il CORS. Un browser permette agli script in una pagina web di accedere ai dati contenuti in un'altra pagina web solo se hanno la stessa origine.

Viene implementato inviando degli header HTTP in req/res. Ci sono le richieste semplici e le richieste preflight.
Per le richieste semplici sono ammessi i metodi: GET, HEAD, POST, sono ammessi gli header: accept, acceptlanguage, content-language, content-type alcuni valori per l'header content-type.
Per le richieste complesse: il browser invia una richiesta preliminare per verificare se il server accetta la richiesta, quest'ultimo dovrà rispondere con intestazioni appropriate. Fa parte di questa categoria *put*, che è utilizzata per aggiornare risorse su un server.

Nelle richieste preflight fanno precedere alla richiesta principale una richiesta preventiva che si chiama OPTIONS (per chiedere al server quali metodi supporta per il file in questione), quindi in totale si fanno 2 richieste.

Un esempio di richiesta cross-origin: il codice JavaScript front-end servito da `https://domain-a.com` usa `fetch()` per effettuare una richiesta per `https://domain-b.com/data.json`.

Metodi HTTP

I metodi HTTP più comuni sono GET e POST. Il metodo GET è usato per ottenere il contenuto della risorsa che viene indicata nell'endpoint. Mentre il metodo POST è usato per inviare informazioni al server, ad esempio i dati di un form.

Altri metodi sono GET, POST, PUT, DELETE, PATCH.

NodeJS

Può essere considerato come un ambiente runtime per JavaScript costruito sopra il motore V8 di Google.

- Ci fornisce un contesto dove possiamo scrivere codice JavaScript su qualsiasi piattaforma dove Node.js può essere installato
- L'ambiente ideale dove usare node.js è il server

Architettura di Nodejs com’è fatta, quanti thread ha? Ha le code?

Node.js funziona così:

1. Per servire le richieste, Node.js mantiene un pool di thread limitato.
2. Ogni volta che arriva una richiesta, Node.js la mette in una coda.
3. Ora entra in scena il “ciclo di eventi” a thread singolo – il componente core. Questo ciclo di eventi aspetta le richieste a tempo indeterminato.
4. Quando arriva una richiesta, il ciclo la prende dalla coda e controlla se questa richiede un’operazione di input/output (I/O) bloccante. In caso contrario, elabora la richiesta e invia una risposta.
5. Se la richiesta deve eseguire un’operazione bloccante, per elaborare la richiesta il ciclo dell’evento assegna un thread dal pool di thread interni. I thread interni disponibili sono limitati. Questo gruppo di thread ausiliari è chiamato worker group.
6. Il ciclo degli eventi tiene traccia delle richieste bloccanti e le mette in coda una volta che il task bloccante è stato elaborato. In questo modo mantiene la sua natura non bloccante.

Moduli

Un pezzo di codice utilizzabile che racchiude i dettagli dell'implementazione.

- I moduli possono essere utilizzati l'uno con l'altro e utilizzano speciali direttive per esportare e importare funzionalità.
- Moduli inJS
  - AMD - uno dei più antichi sistemi di moduli, inizialmente implementato dalla libreria require.js.
  - CommonJS - il sistema di moduli creato per il server Node.js.
  - ES6 Moduls - sistema di moduli a livello di linguaggio apparso nello standard nel 2015.
- Core Modules
  - installati di sistema con node
- Local Modules
  - li creiamo localmente
- Moduli di terze parti
  - li installiamo con *npm*

Core Module	**Description**
http	Il modulo http include classi, metodi ed eventi per creare il server http di

Core Module	Description
	Node.js.
url	Il modulo url include metodi per la risoluzione e l'analisi degli URL.
querystring	Il modulo querystring include metodi per gestire le stringhe di query.
path	Il modulo path include metodi per gestire i percorsi dei file.
fs	Il modulo fs include classi, metodi ed eventi per lavorare con l'I/O dei file.
util	Il modulo util comprende funzioni di utilità utili per i programmatori.

Routing

Per Routing si intende determinare come un'applicazione risponde a una richiesta client a un endpoint particolare, il quale è un URI (o percorso) e un metodo di richiesta HTTP specifico (GET, POST e così via).

Esempio:

```
app.get('/', function(res, req) {
        res.send("GET request to the homepage")
})
```

Express

Express è il framework web Node.js più popolare, ed è la libreria sottostante per un certo numero di altri framework Node.js popolari. Fornisce meccanismi per:

- Scrivere gestori per richieste con diversi verbi HTTP su diversi percorsi URL (route).
- Integrare con i motori di rendering "view" per generare risposte inserendo dati nei modelli.
- Imposta le impostazioni comuni delle applicazioni web, come la porta da utilizzare per la connessione e la posizione dei modelli utilizzati per il rendering della risposta.
- Aggiungere un ulteriore "middleware" di elaborazione delle richieste in qualsiasi punto della pipeline di gestione delle richieste.

REST (Representational State Transfer)

- insieme di linee guida o principi per la realizzazione di una architettura di sistema
- uno stile architetturale
- non si riferisce ad un sistema concreto

- non si tratta di uno standard

Ha 5 principi:

1. Identificazione delle risorse
2. Utilizzo esplicito dei metodi HTTP
3. Risorse autodescrittive
4. Collegamenti tra risorse
5. Comunicazione senza stato

Operazioni CRUD

Acronimo per:
create, read (aka retrieve), update, and delete

- Operazioni di base che posso fare su una risorsa
  - Create (creare una risorsa)
  - Read o Retrieve (leggere una risorsa)
  - Update (aggiornare una risorsa)
  - Delete (eliminare una risorsa)

Risorsa	**GET** **read**	**POST** **create**	**PUT** **update**	**DELETE**
/books	Ritorna una lista di libri	Crea un nuovo libro	Aggiorna i dati di tutti i libri	Elimina tutti i libri
/books/145	Ritorna uno specifico libro	metodo non consentito (405)	Aggiorna uno specifico libro	Elimina uno specifico libro

Comunicazione Stateless

- comunicazione stateless: ciascuna richiesta non ha alcuna relazione con le richieste precedenti e successive
  - La responsabilità della gestione dello stato dell'applicazione non deve essere conferita al server, ma rientra nei compiti del client.
  - La principale ragione di questa scelta è la scalabilità: mantenere lo stato di una sessione ha un costo in termini di risorse sul server e all'aumentare del numero di client tale costo può diventare insostenibile.
  - Inoltre, con una comunicazione senza stato è possibile creare cluster di server che possono rispondere ai client senza vincoli sulla sessione corrente, ottimizzando le prestazioni globali dell'applicazione.

Parlami delle API cosa fai dal punto di vista del server per fare richieste agli altri siti?

Le API (Application Programming Interfaces) sono interfacce che permettono alle applicazioni di parlare con altre applicazioni, permettono di inviare e ricevere dati tra server e client.

Il processo per effettuare una richiesta API è:

1. Decidere che tipo di richiesta HTTP fare se GET, POST, PUT, DELETE
2. Costruzione della URL
3. Invio della richiesta
4. Ricezione risposta: server riceve da un server esterno, in formato JSON o XML, contenente i dati richiesti o un messaggio di errore.
5. Gestione della risposta: viene elaborata la risposta, se ricevuta correttamente(stato 200), si prosegue con elaborazione dati. Se errori (stato 404) si gestiscono gli errori.

API REST

Le API REST comunicano tramite richieste HTTP per eseguire funzioni di database standard come la creazione, lettura, aggiornamento e eliminazione di record (insieme di operazioni noto anche come CRUD) all'interno della risorsa.

Ad esempio, un'API REST utilizzerà una richiesta GET per recuperare un record. Una richiesta POST crea un nuovo record. Una richiesta PUT aggiorna un record e una richiesta DELETE ne elimina uno. Tutti i metodi HTTP possono essere utilizzati nelle chiamate API. Un'API REST ben progettata è simile a un sito web in esecuzione in un browser web con funzionalità HTTP integrata.

Con REST, vengono identificate delle risorse, ossia degli aggregati di informazioni che il servizio web offre. Se ad esempio parliamo di un e-commerce che vuole rendere pubblici i prodotti in vendita e i relativi prezzi potremmo individuare come risorse il singolo prodotto, una lista di prodotti selezionati, etc. Tali risorse vengono condivise in Rete mediante protocollo HTTP e proprio qui sta l’idea geniale alla base di REST: al suo interno HTTP possiede già tutto ciò di cui uno scambio di informazioni necessita pertanto non c’è bisogno di creare alcuna ulteriore sovrastruttura. Sfruttando gli elementi interni a questo protocollo potremo instaurare un dialogo tra il client ed il server.

Il protocollo HTTP funziona mediante un modello richiesta/risposta. Un client invia una richiesta al server, il server risponde. La richiesta può includere al suo interno una serie di funzionalità. Si può chiedere solo di leggere dati, di modificarli, crearli o cancellarli. Tutto ciò viene specificato mediante un campo della richiesta HTTP detto metodo. In REST si usano per lo più quattro metodi che indicheranno quali di queste operazioni il client starà

richiedendo al server: con il metodo GET si chiederà solo di leggere dei dati, con il POST di crearne di nuovi, con il PUT di modificarli e con il DELETE di cancellarli.

Esempio API GET:

```
app.get('/api/v1/products/:id', (req, res) => {
    console.log(req.params);

    const prod = products.find((el) => el.id == req.params.id);
    console.log(prod);

    if (prod == undefined) {
        res.status(404).json({
            status: 'fail',
            message: 'ID non trovato'
        });
    } else {
        res.status(200).json({
            status: 'success',
            data: {
                product: prod,
            },
        });
    }
});
```

Esempio API POST:

```
app.post('/api/v1/products', (req, res) => {
    const newId = products[products.length - 1].id + 1;
    const newProd = Object.assign({ id: newId }, req.body);

    products.push(newProd);
    res.status(201).json({
        status: 'success',
        data: { product: newProd },
    });
});
```

Sapresti fare un middleware, che callback gli passi?

Una funzionalità di Express è rappresentata dai **Middleware**. Si tratta di semplici funzioni che possono essere usate all'interno dell'applicazione. Le funzioni Middleware hanno accesso all'oggetto richiesta (solitamente denominato req), all'oggetto risposta (res) e a una

funzione callback che viene solitamente denominata next. Le funzioni Middleware sono solitamente disposte in cascata. Per fare in modo che una passi il controllo alla funzione successiva è necessario invocare la funzione next(). Solitamente, se non viene inviata una risposta al client, si invoca la funzione next() per fare in modo che le altre funzioni Middleware lungo la catena possano elaborare la richiesta e passarla eventualmente alla funzione successiva.

Esempio:

```
app.use(function (req, res, next) {
    console.log('Hello from the middleware!');
    next();
});

app.use('/api', function (req, res, next) {
    console.log('This middleware handles the data route');
    next();
});
```

Quali campi form conosci?

Il tag `<input>` usato per creare diversi controlli in un form HTML. È un elemento in linea e riceve attributi come `type`.
Ad esempio come

```
<input type="text" placeholder="Enter name" />
```

Oppure

```
<input type="button" value="Submit" />
```

questo input crea un bottone, che può essere manipolato in base all'utilizzo.

Head e Header

La differenza è che il primo contiene i metadata e il secondo è un segmento di pagina e si trova nel body.

Form

- **HTTP GET**, è il metodo con cui vengono richieste la maggior parte delle informazioni ad un Web server, tali richieste vengono veicolate tramite query string, cioè la parte di un URL che contiene dei parametri da passare in input ad un'applicazione - si usa ? all'inizio e & fra i dati, ad esempio: `www.miosito.com/pagina-richiesta?id=123&page=3`
- **POST**, i dati vengono inviati nel body della richiesta.