Bug | Request

Tecnologie & Linguaggi Web

Last Update : 13/11/2023

- Responsive Web Design
- HTML
  - Tags
    - Tag `<div>`
  - Browser
- XHTML
- CSS
- Bootstrap
- XML
- JSON
- JSON Vs XML
- REST
- JavaScript
  - Variabili e Scope
  - DOM
    - HTML DOM
  - Eventi
  - Promises
  - Fetch
- NodeJS
  - NPM
- SwaggerJS
- MongoDB
- Downloads

Responsive Web Design

Il responsive web design riguarda la creazione di siti web che si adattano automaticamente per avere un aspetto adatto a seconda del dispositivo in cui vengono visualizzati, dai piccoli telefoni ai desktop di grandi dimensioni.

HTML

→ Sintassi HTML ←

Hyper**T**ext **M**arkup **L**an Viene usato per la creazione delle pagine Web

- È uno **standard** per la rappresentazione
- Descrive la **struttura della pagina**
- **Linguaggio di markup** (basato sui **tag**), ossia un linguaggio per strutturare e marcare i documenti in maniera indipendente dall'applicazione, modellato per rendere esplicita una particolare interpretazione di un testo.
- Descrive **dati** e **regole** su come mostrare i dati
- Ha poche e semplici regole sintattiche
- Non ha nessun meccanismo di decisione o iterazione (linguaggio di markup e non di programmazione)

> La struttura dei documenti HTML è divisa in due parti: testa (**head**) e corpo (**body**)

Scheletro Pagina HTML

```
<!DOCTYPE html>
<html lang="it">
   <head>
      <meta charset="utf-8">
   </head>
</html>
```

Tags

La struttura delle pagine web è definita attraverso dei **tag** ossia dei marcatori che identificano porzioni di testo, e personalizzano la pagina.
Tutto il codice contenuto all interno del tag `<html>` è codice html.
Lo **head** contiene informazioni/metadati relativi al documento (non immediati), ad esempio descrive come il documento deve essere letto e interpretato ed eventualmente il titolo Il **body** contiene il documento vero e proprio, presentando il contenuto in diversi modi sia testo ma anche immagini, video, audio.
Il tag **div** che è un blocco contenitore generico, fornisce un vero e proprio elemento strutturale della pagina, perchè suddivide gli spazi della pagina in zone in modo seplice e dettagliato.
Altri tag es img, table, ul/li, ol, li, form, input, a, select/option.

Esempio di tag: `<nome_tag attributi>contenuto</nome_tag>`

Tag `<div>`

Un **tag** molto utilizzato è `<div>` che rappresenta un blocco che funziona da contenitore. È utile sopratutto nella creazione del layout della pagina in quanto ne permette di definire la struttura.
Suddivide gli spazi in zone per progettare il sito in modo semplice e dettagliato

Esempio di struttura con *tag div*

```
<div id="container">
   <div id="header">
      <div id="navigation"></div><!--#navigation-->
   </div><!--#header-->
   <div id="main"></div><!--#main-->
   <div id="sidebar"></div><!--#sidebar-->
   <div id="footer"></div><!--#footer-->
</div><!--#container-->
```

Un tratto che ha reso HTML popolare sono gli ipertesti (o **link**), ovvero un collegamento tra un testo e un altro.
I **link** sono formati da due componenti: contenuto e risorsa a cui si riferiscono.
Esempio e sintassi dei *link*: `<a href="remote_or_local_address">Testo</a>`
La risorsa a cui i link fanno riferimento possono essere: immagini, documenti, altre pagine HTML, file, e-mail, numeri ecc... (è possibile utilizzare anche il *relative path*)

> I **Commenti** in HTML → `<!-- commento -->`
> **HTML** è **case unsensitive** → `<HEAD></HEAD>` == `<head></head>`
> **XHTML** è **case sensitive** → `<HEAD></HEAD>` != `<head></head>`
> È comunque consigliabile utilizzare il carattere minuscolo

Browser

Un **browser** interpreta e legge i documenti HTML, e serve per la navigazione web, infatti esso richiede risorse attraverso il web e le mostra nella finestra.
Inoltre contiene alcuni **tool per gli sviluppatori** per esempio il debug della pagina, analisi di sorgenti e stili, accesso allo storage e cookie e molto altro, alcuni browser famosi sono: *Chrome*, *Firefox*, *Opera*, *Safari*.

Composizione del Browser:

- **Layout Engine**
  riceve input dal browser (*Url bar*, *search box*, *mouse clicks* e *key presses*) e li passa al rendering engine
- **Rendering Engine**
  riceve il codice HTML e lo interpreta mostrandolo a video
- **User Interface**
  controlli del browser, ad esempio pulsanti per andare avanti/indietro, bookmarks ecc..
- **JavaScript Engine**
  engine che effettua il parse ed esegue il codice JavaScript per poi restituirne il risultato
- **Network Layer**
  gestisce le funzioni di rete: *criptazione*, *richieste http* e le varie configurazioni...
- **Storage**
  porzione di memoria dove il browser memorizza *cached file*, *cookie* e oggetti creati con JavaScript
- **Operating System Interface**
  componente che interagisce con il Sistema Operativo per disegnare i diversi e elemnti e gestire la finestra

---

XHTML

E**X**tensible **H**yper**T**ext **M**arkup **L**anguage

Linguaggio di markup quasi identico a html, ma più stringente, supportato dai principali browser, motivo per cui si usa è che molte pagine web contengono html errato, dove per interpretare codice sbagliato richiede

maggiori risorse o potenza computazionale, quindi è un **linguaggio di markup corretto** dove i documenti devono essere ben formati (es tag sempre chiusi, innestati correttamente).

Come converto un file html in x-html?

- aggiungendo un XHTML `<!DOCTYPE>`; nella prima linea di ogni pagina,
- aggiungendo l'attributo xmlns all'elemento html `<html xmlns="http://*.w3.org/*/xhtml"></html>`
- cambiando tutti i nomi di elemento in minuscolo etc…

CSS

→ Sintassi CSS ←

Cascading **S**tyle **S**heet

File css non sono altro che **fogli di stile a cascata**, è uno dei linguaggi fondamentali del W3C (organizzazione che fornisce linee guida e regole per la specifica delle pagine), serve per **associare uno stile al testo delle pagine**, riducendo il carico del lavoro, permette inoltre di controllare il layout delle pagine web multiple in un colpo solo, possono essere separate dalla pagina html.
Pagine più leggere e facili da modificare.

È un **insieme di regole**, ogni regola è formata da un **selettore** associato ad un blocco delle dichiarazioni, ossia delle coppie **propietà**, **valore**.
Selettore per i tag (tag{}), selettore per id (`#id{}`), selettore per le classi (.classe{})
Per inserire css **esterni** uso `<link>` nello `<head>`, per quelli **interni** metto tutte le regole nel tag `<style>`, oppure **in linea** con l'attributo `style=""`.

"a cascata" perché è il modo in cui vengono valutate le regole css.
Infatti lo User agent deve applicare un algoritmo per stabilire i valori di una combinazione di elementi e proprietà. **Questa valutazione a cascata avviene in 4 fasi**:

1. Per prima cosa bisogna selezionare tutte le dichiarazioni che applicano a un elemento/proprietà per il media type richiesto
2. Il primo ordinamento è fatto in base a peso e origine
3. Il secondo ordinamento è fatto in base alla specificity
4. Il terzo ordinamento è fatto in base all'ordine

Css box model: ossia tutti gli elementi html sono delle box, da più esterno al più interno abbiamo: **margin** area trasparente attorno al bordo, **border** un bordo che circonda padding e content , **padding** area trasparente attorno al content, il **content** dove risiedono testo e immagini.

Bootstrap

→ Ulteriori info nel sito ufficiale ←

Bootstrap è il framework HTML, CSS e JavaScript più popolare per lo sviluppo di siti Web responsive e mobile-first. **Bootstrap** è un framework front-end gratuito per uno sviluppo web più rapido e semplice, include modelli di progettazione basati su HTML e CSS per molti elementi, oltre a plug-in JavaScript opzionali, inoltre ti dà anche la possibilità di creare facilmente **design reattivi**, ossia design che si adattano automaticamente per avere un bell'aspetto su tutti i dispositivi, dai piccoli telefoni ai desktop di grandi dimensioni.
In pratica consiste in una libreria open-source per realizzare siti web e applicazioni web responsive.
Per usarlo basta includere i fogli di stile e gli script.
Incorpora alcuni elementi come: **grid**, **table**, **buttons** e molto altro.

I vantaggi dell'utilizzo di Bootstrap sono:

- Facile utilizzo
- Funzionalità reattive
- Approccio mobile-first
- Compatibilità con tutti i browser moderni

Per includere Bootstrap è necessario inserire i seguenti elementi all'interno del progetto:

CSS

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/${VERSION_NUMBER}/css/bo
```

JavaScript

```
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/${VERSION_NUMBER}/js/bootstrap.min.js"><
```

jQuery Library

```
<script src="https://code.jquery.com/jquery-latest.js"></script>
```

Per garantire il corretto rendering e zoom al tocco è necessario utilizzare con Bootstrap il tag `<meta>`, attraverso il codice:

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

- `width=device-width` → imposta la larghezza in modo da seguire la larghezza del display del device in uso
- `initial-scale=1` → imposta il livello di zoom iniziale quando la pagina viene caricata la prima volta

Info su Bootstrap

> Se l'utente ha già visitato un sito che utilizza Bootstrap allora non verrà effettuato nuovamente il download, ma verrà caricato dalla cache, velocizzando i tempi di caricamento. Inoltre Bootstrap è creato in modo che quando un utente richieda un file, esso venga fornito dal server più vicino per ridurre ulteriormente i tempi di caricamento.

XML

→ Sintassi XML ←

E**X**tensible **M**arkup **L**anguage

Nato dalla necessità di un nuovo e comune formato dei dati per Internet, necessità di regole semplici comuni e semplici da comprendere, capacità di definire una struttura delle informazioni per diversi domini, formato abbastanza formale, e leggibile.

XML è uno strumento per trasmettere informazioni, indipendente dalla piattaforma, dal software e dall'hardware.
Ossia **un linguaggio di markup** come HTML, dove i tag di XML non sono predefiniti, creato per strutturare e memorizzare e trasmettere dati.
Le informazioni sono collezionate in unità chiamate **documento XML**, il quale contiene uno o più elementi dove ogni elemento ha un nome, ossia all'interno dei tag abbiamo le informazioni.
XML non vuole sostituire HTML ma la tendenza è quella di rappresentare i dati con XML e mostrarli con HTML.
Infatti il solo scopo di XML è quello di strutturare, memorizzare e trasmettere i dati.
Inoltre con xml **è possibile scambiare i dati fra sistemi incompatibili**.

Sintassi

- in XML è illegale omettere il tag di chiusura
- i nomi dei tag XML sono case-sensitive
- è presente l'annidamento dei tag
- tutti i documenti XML devono avere un nodo radice
- tutti gli altri elementi devono essere compresi nell'elemento radice
- tutti gli elementi possono avere dei nodi figli
- i valori di tutti gli attributi devono essere tra doppi apici
- evito di usare gli attributi se possibili li uso solo per le istruzioni di controllo

```
<?xml version="1.0" encoding="utf-8"?>
<event>
   <date>20 Settembre 2023</date>
   <priority>High</priority>
   <name>Esame Tecnologie e Linguaggi Web</name>
   <note>Presentazione e discussione orale del progetto</note>
   <url>https://ariel.unimi.it</url>
</event>
```

Validazione XML e **DTD**

Un documento XML sintatticamente corretto è detto **ben formato**.
Un documento XML che rispetta uno schema (o un DTD) è detto **valido**.
Per leggere, modificare o creare un documento XML è necessario un **parser XML** (js con `DOMParser()`).

JSON

→ Sintassi JSON ←

Java**S**cript **O**bject **N**otation

È un **text-based open standard** leggero disegnato per lo scambio di dati human-readable.
Usato quando si scrivono applicazioni basate su JavaScript (include estensioni al browser e siti web);
sopratutto usato per **trasmettere dati tra server e applicazioni web** (servizi **API**).
Facile da usare da leggere e scrivere, molto leggero, ed indipendente dal linguaggio.

La sintassi JSON è considerata come un sottoinsieme della **sintassi JavaScript**,
infatti dati rappresentati come coppia chiave, valore, le parentesi graffe contengono oggetti,
le parentesi quadre definiscono array.

Esempio JSON

```
{
   "exams":[
      {
         "id": "01",
         "name": "Tecnologie e Linguaggi Web",
         "teacher": "V.B.",
         "year": 3,
         "url": "https://ariel.unimi.it/tlw",
      },
      {
         "id": "02",
         "name": "Programmazione II",
         "teacher": "M.S.",
         "year": 2,
         "url": "https://ariel.unimi.it/prog2",
      }
   ]
}
```

> I **tipi di dato** supportati da JSON sono:
> *Number*, *String*, *Boolean*, *Array*, *Value*, *Object* (**key**:**value**), *Whitespace*, *null*

JSON Schema
È una specifica per la definizione dei dati JSON, serve
per validare JSON, con diverse librerie per i vari linguaggi (validatore più valido → **JVS**)

> Altre info di JSON e implementazioni nei vari linguaggi.

JSON Vs XML

JSON è simile a XML ma **più leggero**.
Sono entrambi formati human readable e indipendenti dal linguaggio.
Ogni valore in XML richiede apertura e chiusura di un tag, in **JSON solo il valore**.

Si possono confrontare su 3 fattori:

1. **verbosità**: xml più verboso di JSON che è più veloce da scrivere
2. **utilizzo degli array**: xml usato per dati strutturali che non contengono array, invece json li include
3. **parsing**: in JS molto semplice parsare json (`JSON.parse or JSON.stringify`), chiunque può parsare XML

REST

REST → **RE**presentational **S**tate **T**ransfer

- Gestisce un tipo di contenuto variabile che può essere XML, ma principalmente JSON.
- **è un'astrazione di elementi architetturali** in un sistema hypermedia distribuito
- **ignora i dettagli implementativi** dei componenti e la **sintassi dei protocolli**
- si focalizza sul ruolo dei componenti

Data Element	Modern Web Example
resource	conceptual target of hypertext reference
resource identifier	URL, URN (**U**nified **R**esource **N**ame)
representation	HTML document, JPEG image
representation metadata	media type, last-modified time
resource metadata	source link, alternates, vary
control data	if-modified-since, cache-control

La natura e lo stato dei **data element** è un aspetto essenziale di REST, le componenti REST trasferiscono una risorsa usando una delle rappresentazioni standard.
Ogni **resource** è una membership function che al tempo t viene associata a un set di entità/valori.

Resource identifier

l'autore sceglie l'identificatore, due uri possono puntare alla stessa risorsa.

Addressability

una applicazione è indirizzabile se espone i suoi dati come risorse, espone un uri per ogni info di interesse.

Statelessness

ogni richiesta HTTP è eseguita in isolation, infatti il server non usa informazioni da richieste precedenti, il client inserisce tutto ciò che è necessario a soddisfare la richiesta, lo stato renderebbe le richieste molto più semplici ma la gestione del client molto più complicata.

Representation

usata per ottenere lo stato di una risorsa, trasferita tra i componenti, sequenza di dati più byte che la descrivono, il client sceglie la rappresentazione, e il server sceglie la rappresentazione in base allo header della richiesta (xml, json etc…).

Uniform Interface per tutte le risorse interfaccia comune

- `GET` ottiene una rappresentazione della risorsa
- `POST` crea una nuova risorsa
- `PUT` crea o aggiorna una risorsa
- `DELETE` cancella una risorsa

Uniform Interface Principle: **CRUD** example

CRUD	REST	*action*
CREATE	**P**OST	*create a sub resource*
READ	**G**ET	*retrieve the current state of the resource*
UPDATE	**P**UT	*initialize or update the state of a resource at the given URI*
DELETE	**D**ELETE	*clear a resource, after the URI is no longer valid*

> `POST` è usata per creare risorse subordinate, `PUT` per creare/modificare risorse
> Con la **PUT** il client decide il nome della risorsa, con la **POST** lo decide il server.

RESTful-Services

Un servizio basato su REST è chiamato **RESTful service**, **REST non dipende da nessun protocollo**, anche se quasi tutti i RESTful service usano HTTP.
Visione astratta REST client:

- definisco le info da aggiungere alla HTTP request (header, body, method, uri)
- creare l'HTTP request e mandarla al server HTTP
- parsare la risposta e fornire le info al codice REST

Il client usa XMl parser per interpretare la risposta, client REST possono essere sviluppati in molti linguaggi (js).

Visione astratta REST server:

- **accettare** la HTTP request
- **parsare** la HTTP request ed estrarre le info importanti
- **spedire** la risposta al client

REST accede ai dati usando **URI**.
Safety: richieste di sola lettura che non causano danni (GET, HEAD).
Idempotenza: L'esecuzione di un commando una o più volte ha lo stesso effetto (GET, HEAD, PUT, DELETE).

JavaScript

→ Sintassi JavaScript ←

Linguaggio di scripting, tipo particolare di linguaggio di programmazione, utilizzato appunto per scrivere **script**.

> **Script**: È un programma che inseriamo nella pagina web

- È formato da un insieme di istruzioni, essenziale per la visualizzazione di **pagine web dinamiche**, ossia pagine web dove il contenuto viene generato al momento della richiesta/visualizzazione
- È un **linguaggio lato client**: viene elaborato tramite il **client** (browser), senza la necessità di un server per poter visualizzare la pagina
- È un **linguaggio interpretato**, ossia che il sorgente non deve essere compilato per essere eseguito, inoltre l'interprete di JS è contenuto all'interno del browser.

JavaScript inoltre è:

- case-sensitive
- weak-typed
- event-driven
- Object-based language

Il codice JS può essere incluso in un tag `<script>codice JavaScript</script>`, oppure semplicemente indicando il percorso del file JS `<script src="path_to_file.js"></script>`

Possibilità di accesso agli oggetti tramite **nodi** (*ex*: aggiungo con `appendChild()`)

Oggetti JS: è una collezione di dati dotata di nome, posso accedere alle sue propietà con la **dot-notation**, oggetto JS costruito tramite costruttore.

Variabili e Scope

Scope:

- **Globale** → per le variabili definite fuori da funzioni
- **Locale** → per le variabili definite esplicitamente all'interno di funzioni

> ATTENZIONE un blocco NON delimita uno scope!

```
x = '3' + 2; // string "32"
{
 { x = 5 }    // internal block
 y = x + 3;  // x = 5 and not "323" -> y = 8
}
```

DOM

Document **O**bject **M**odel

È uno dei concetti chiave di *JS*, è uno standard di W3C, costituito da tre parti:

- **Core DOM**: modello standard per tutti i tipi di documenti
- **XML DOM**: modello standard per documenti XML

- **HTML DOM**: modello standard per documenti HTML

HTML DOM

Definisce uno standard su come come ottenere, cambiare, aggiungere o modificare elementi HTML.
JavaScript usa **HTML DOM** per modificare tutti gli elementi di una pagina web.
In pratica definisce ogni elemento HTML come oggetto, che ha delle proprietà e dei metodi per accedervi, oltre a creare degli eventi per gli elementi stessi.

> quando una pagina è caricata il browser crea il DOM della pagina, generato secondo una gerarchia.

Tra gli oggetti del DOM ho:

- `window` che è la finestra corrente del browser
  - `document` il contenuto della finestra
  - `navigator` oggetto relativo al browser
  - `location` oggetto relativo alle informazioni sull'indirizzo della risorsa, il percorso etc...
  - `history` oggetto relativo alla navigazione delle pagine (avanti/indietro)

> *Come accedo agli elementi della pagina attraverso id?* `document.getElementByID(id)`

Eventi

I programmi JavaScript sono tipicamente “guidati dagli eventi” (**event-driven**).
Gli eventi sono scatenati da azioni dell’utente sulla pagina Web.
Un programma JavaScript deve contenere un gestore di eventi (**event handler**),
che sia in grado di ricevere e interpretare le azioni dell’utente (eventi).
Il DOM fornisce una serie di gestori di eventi predefiniti.
Per esempio evento di `onClick` , `onLoad` , `onMouseOver` , inoltre questi eventi possono chiamare funzioni JS che raccolgono i dati dal form.

```
<!-- L'utente fa click (onClick) sul link <a ..> -->
<a href="#" onClick="window.print()">

<!-- L'utente invia (onSubmit) il modulo <form ..> -->
<form name="module" onSubmit="alert('Sent!');">

<!-- L'utente porta il cursore (onFocus) nel campo di testo <input ..> -->
<input type="text" name="login" onFocus="js_action();">

<!-- L'utente carica (onLoad) la pagina <body ..> -->
<body onLoad="alert('Loaded!');">
```

Promises

Una **promise** è un particolare oggetto che connette il **codice produttore**, una qualsiasi operazione che richiede tempo (*come il download di file da server*),
con il **codice consumatore**; cioè un operazione che necessita del risultato del codice produttore, rendendo il risultato disponibile al consumatore quando il codice produttore è terminato.

```
let promise = new Promise(function(resolve, reject) {
    // codice produttore
});
```

I due argomenti della funzione sono delle callback che vengono eseguite quando la promise è risolta o rifiutata:

- **resolve(value)** - se il processo termina correttamente con il valore passato come parametro
- **reject(reason)** - se il processo termina con un errore

L'oggetto **Promise** ha due *proprietà interne*:

- **state**
  - **pending** stato iniziale
  - **fulfilled** se viene invocato resolve
  - **rejected** se viene invocato reject
- **result**
  - **undefined** stato iniziale
  - **value** se viene invocato resolve
  - **error** se viene invocato reject

Esempio di utilizzo

```
var promise = new Promise(function(resolve, reject) {
   // dopo 1 secondo segnala che l'istruzione ha avuto successo
   setTimeout(() => resolve("done!"), 1000);
})

// utilizzo della promise
promise.then(function(valore) { alert(valore) });

// versione compatta
promise.then(valore => alert(valore));
```

Fetch

JavaScript può **inviare richieste di rete al server** e caricare nuove informazioni ogni volta che è necessario.
Il metodo `fetch()` è tra tutti il più moderno e versatile.
Sintassi di base: `var promise = fetch(url, [options])`, senza options è una GET, la funzione restituisce una promise che verrà gestita come precedentemente detto, oppure con `await`.
Posso anche specificare un method diverso ad esempio POST con anche il body.

Ottenere una riposta avviene in due fasi:

- **Valutazione dello stato**

```
let response = await fetch(url);
if (response.ok) { // se HTTP-status è 200..299
```

```
    var json = await response.json();
} else {
    alert("HTTP-ERROR: " + response.status);
}
```

- **Risposta**

  fornisce diversi metodi per accedere al body in formati differenti

  - `response.text()` legge la risposta e restituisce un testo
  - `response.json()` interpreta la risposta e restituisce la response come JSON
  - `response.formData()` restituisce la response come oggetto FormData
  - `response.blob()` restituisce la response come Blob (binary large object), dato binario con type

```
fetch("https://server.com/api/movie/")
    .then(response => response.json())
    .then(results => alert(results.results[0].title())
```

> Per eseguire una richiesta `POST` o con un altro metodo, utilizziamo le opzioni di fetch

```
let user = { nickname: "K", id: "01" }
// var promise = fetch(url, [opts])
var response = await fetch("server/path", {
    method: 'POST',
    headers: { 'Content-Type': 'application/json;charset=utf-8' },
    body: JSON.stringify(user)
})
```

NodeJS

Node.js è un ambiente JavaScript multipiattaforma **lato server**, basato su motore JS, inoltre fornisce moduli per interagire con le risorse del sistema.
I programmi Node.js possono essere creati utilizzando file JavaScript che devono essere eseguiti tramite Node.js.

Semplice script Node, in file `app.js`

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
```

```
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Nella Shell

```
node app.js
# open browser at location:
{open,xdg-open} http://localhost:3000
```

NPM

NPM → **N**ode.js **P**ackage **M**anager
È il gestore dei pacchetti di NodeJS; fornisce un repository pubblico di pacchetti, uno strumento a riga di comando per lavorare con i pacchetti.

I pacchetti possono essere installati a livello **globale** (accessibili da tutto il sistema),
oppure a livello **locale**, memorizzati nella cartella selezionata nella sottocartella `node_modules` e disponibili esclusivamente al progetto residente in quella cartella.

Comandi Principali NPM

```
# Inizializza e configura node per la cartella dove viene eseguito
# creando anche il file <package.json> che contiene metadati e info sul progetto oltre
# a dipendenze dell'applicazione e utili per lo sviluppo
npm init

# installa il pacchetto 'express' a livello globale
npm install --global express
# installa il pacchetto 'express' a livello locale
npm install express
# installa il pacchetto 'express' a livello locale identificandolo come dipendenza della
# applicazione (flag '-S' o '--save')
npm install -S express
# installa il pacchetto 'express' a livello locale identificandolo come dipendenza dello
# sviluppatore (flag '-D' o '--save-dev')
npm install -D express
# rimuove il pacchetto 'express'
npm uninstall express
```

SwaggerJS

Swagger ti consente di descrivere la struttura delle `API` in modo che le macchine possano leggerle.
In pratica è un *tool* utile per la creazione della documentazione delle API di un progetto e del loro utilizzo.
A seconda dell'applicazione, lo swagger consiste in un file di testo (con formato JSON o YAML).
L'interfaccia utente per semplificare la documentazione è basata su HTML e JavaScript.

> Per lo sviluppo delle API è essenziale una documentazione adeguata e comprensibile.
> Senza la documentazione è difficile per gli sviluppatori, utilizzare le interfacce.
> Ciò è vero soprattutto per le API di dominio pubblico!

Utilizzare Swagger è attualmente il modo migliore per documentare le **API REST**,
poiché è in grado di mappare quasi tutti i servizi web e le informazioni relative all'interfaccia.
Swagger evolve con il sistema e tutte le modifiche apportate vengono documentate automaticamente.

È possibile utilizzarlo con il pacchetto `express` attraverso l'apposito modulo disponibile tramite npm.

Esempio utilizzo

```
// installare prima il pacchetto tramite comando shell
//   npm install swagger-ui-express

// nel file app.js
const express = require('express');
const app = express();
const swaggerUi = require('swagger-ui-express');
const swaggerDocument = require('./swagger.json');

app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocument));
```

MongoDB

→ Sintassi Mongo ←

→ Docs ←

È un database open source (scritto in *C++*) **NoSql senza schema**, composto da collezione di documenti.
Utilizza BSON, "**B**inary J**SON**", al posto di JSON che permette un pò più di dati rispetto a JSON in forma di campo e valore con possibilità di usare delle chiavi esterne, la struttura incorporata riduce la necessità di join.

- Una **collezione** è un insieme di **documenti** di uno stesso tipo
- Può archiviare qualsiasi tipo di dati con qualsiasi dimensione
- Utilizza la RAM per archiviare i dati, questo consente un accesso più rapido ai dati
- È conveniente perché riduce i costi su hardware e archiviazione
- Supporta l'elaborazione di **query** ad-hoc

MongoDB Data Model: è document-based (max *16MB*), documenti in formato BSON, ogni documento è archiviato in una collezione.

MongoDB vs. SQL

MongoDB	SQL
Document	Tuple
Collection	Table/View

MongoDB	SQL
PK: _id Field	PK: Any Attribute(s)
Uniformity not Required	Uniform Relation Schema
Index	Index
Embedded Structure	Joins
Shard	Partition

Altre proprietà di MongoDB:

- È facile da installare, utilizzare → database senza schema
- database senza schema → il codice che creiamo definisce lo schema
- può archiviare qualsiasi tipo di dato con qualsiasi dimensione
- i dati sono archiviati in BSON (key:value), non sono necessari **join** complessi
- può essere replicato e utilizzato su più host
- supporta la proprietà **ACID** (**a**tomicità, **c**oerenza, **i**solamento, **d**urata) per una transazione del database

CRUD
Create
Read
Update
Delete

CRUD permette di inserire dati nel database usando la **shell**

```
show dbs/collection
use DB_NAME
db.DB_NAME # crea un db di nome DB_NAME
# db.<collection/db>.insert(<document/instance>)
db.City.insertOne({"name":"Milan","state":"Italy"})
db.Name.insertMany([{},{}])
```

Considerazioni su Mongo

> database **affidabile**, raccomandato durante la progettazione di un app Web
> **scalabile** che permette di creare database di grandi dimensioni per archiviare grandi quantità di dati, anche di grandi dimensioni, non strutturati.
> È utile per chi cerca un database che abbia un ottima disponibilità, elaborazione rapida, **buon backup**, nessuna perdita di informazioni e oltretutto sia gratuito.

Downloads

- File Markdown
- File PDF
- File HTML

Return to Top