APPUNTI CORSO TECNOLOGIE WEB - 0312212INF01IV

PROF. STEFANO D'URSO

A.A. 2024/25

Sommario

WEB 1.0 (1990-2000)

- 1989-90 CERN, Tim Berners Lee
- NEXUS primo browser
- Read-only web
- *“Se non ci sei, non esisti”*
- Web 1.5: proto-interazione principalmente asincrona(come nei blog)

WEB 2.0 (2000-2006)

- 2004 Web 2.0 Conference, Tim Berners Lee
- Read-write web
- Interazione, condivisione, partecipazione
- “se non ci sei, comunichi la tua assenza”

WEB 3.0 e Web3 (2006-2014)

- Read-write-execute web
- Rete come db, web semantico, assistenti virtuali, responsive design, smart home connection
- Web3 come risposta alla centralizzazione di alcune offerte come i social: blockchain, NFT, Crypto, realtà aumentata, on-demand cloud resources

WEB 4.0

- Spazio e big-data
- Interfacce cervello-computer, IA, Metaverso, IOT

DATI ED INFORMAZIONE

DATO: valore grezzo

INFORMAZIONE: processo di elaborazione del significato del dato grezzo

Dato+elaborazione=Informazione

I sistemi informativi (OPERATIVI: gestiscono il singolo processo e DECISIONALI: gestiscono la gamma dei processi aziendali) guidano la gestione delle informazioni.

INFORMAZIONI E COMUNICAZIONE

Comunicazione: ha bisogno di **mittente** e **destinatario.**

Complesso processo di condivisione tra persone. Il computer è uno strumento di COMUNICAZIONE.

- Regole che disciplinano la comunicazione (protocolli)
- Livelli di comunicazione
- Servizi
- Componenti della rete
- Dispositivi finali ed intermedi
- Architettura della rete

PROTOCOLLO E SERVIZIO

1984: Open System Interconnection, standard dell'ISO (International Standards Organization).

PROTOCOLLO: insieme di regole definite al fine di realizzare quanto due entità **(peer entity)** si sono proposte

SERVIZIO: è un insieme di regole offerte da un livello dello standard al suo livello superiore (da N-1 a N)

SAP (Service Access Protocol): access point che permettono ad un livello di utilizzare i servizi di quello sottostante per la trasmissione ad un'altra entità.

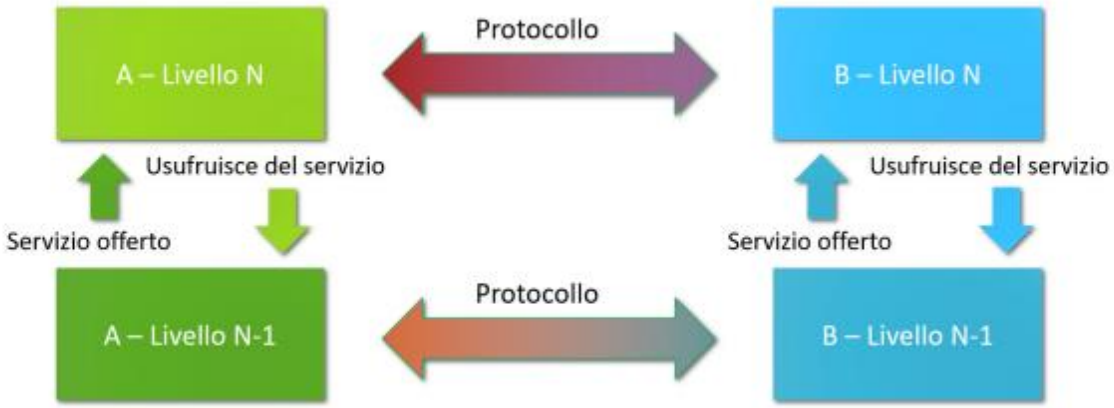


Figura 2: Protocollo e Servizio

Il set di servizi offerto da un livello prende il nome di **interfaccia di livello**. Chi usufruisce di un servizio è noto come **service user**.

IL MODELLO OSI

7 livelli:

1. Application
2. Presentation
3. Session
4. Transport
5. Network
6. Data Link
7. Physical

I primi 3 layer fanno parte della parte end-to-end; il layer di trasporto si occupa di interfacciare i 3 livelli superiori con quelli inferiori. I tre inferiori sono livelli di comunicazione Data Network.

Per trasferire un dato, esso viene frammentato in parti più piccole, dette **data unit.**

Quando il data unit viene trasferito tra due peer entities, prende il nome di **protocol data unit**, quando invece esso viene trasferito da un livello al livello N-1 esso prende il nome di **service data unit**.

Un PDU è diviso in due parti:

- **PCI**: contiene le info di controllo, ed è o **header** o **footer** del DU
- **SDU:** contiene il payload (i dati da trasmettere)

IL MODELLO TCP/IP

4 livelli:

1. **application** (che sostanzialmente racchiude application, presentation e session dell'OSI): fornisce le interfacce tra le applicazioni per accedere alle funzionalità di rete; utilizza protocolli come **http, FTP, SMTP**
2. **transport:** fornisce servizi di trasporto affidabili per le applicazioni; usa protocolli come **TCP/UDP**
3. **Internet** (corrisponde al Network Layer OSI): **ICMP** come protocollo, inoltra i pacchetti sulla rete
4. **Network Access** (corrisponde a Data e Physical OSI): procotolli **ARP e RARP** per risolvere MAC Address e IP

DEVICE FISICI

Livello OSI 1 (Physical):

- Repeater
- Modem
- Hub
  Sostanzialmente, replicano il segnale

Livello OSI 2 (Data Link):

- Switch
- Bridge
  Replicano il segnale e instradano i frame tramite database CAM, che contiene i Mac Address e le porte

Livello OSI 3 (Network):

- Firewall
- Router
  Connettono le LAN con Internet, sono device complessi (hanno processori, ram e memorie), instradano i pacchetti tramite protocolli e tabelle dedicati (RIP, OSPF).

STRUMENTI DI NETWORK TROUBLESHOOTING

- **Ping**
- **Traceroute**
- **IPConfig**
- **NsLookup**
- **Whois**
- **Netstat**

- **Wireshark**

L'EVOLUZIONE TECNICA DI INTERNET

1990, Tim Berners Lee: HTTP (Hypertext Transfer Protocol): protocollo request-response studiato per lo scambio di ipertesta; le risorse vengono identificate univocamente tramite **URL** (Uniform Resourse Locator).

Poi, sempre Tim Berners Lee introduce **l'HTML** (Hypertext Markup Language), un linguaggio di Markup interpretato da browser; è un linguaggio **dichiarativo**, cioè indica caratteristiche senza presentare interattività.

1995, JavaScript: <script>, incluso per la prima volta in Netscape Navigator 2. Permette ai dev di modificare in maniera dinamica e al volo il contenuto di una pagina web.

1996, CSS: linguaggio per supportare il markup degli elementi.

Nascono distinzioni tra Front-end e Back-end developer.

2005, AJAX: una serie di tecnologie che comunicano i background (in modo asincrono), senza interferire con lo stato della pagina (HTML, DOM, JSON o XML, JS); abbiamo una nuova distinzione di:

server-side: si snellisce e si concentra su pure data API o web- service.

Client-side: diventa più complesso e necessita di interi framework per gestire l'interattività con l'utente; l'unica interazione con il server diventa lo scambio dei dati (JSON).

Framework JS: jQuery, Prototype, MooTools

Nascono gli sviluppatori FULL-STACK

FRAMEWORK JS FRONT-END	FRAMEWORK BACKEND
AngularJS	Ruby on Rails (Ruby)
React.js	Django (Python)
Vue.js	Laravel (PHP)
	Node.js

2010, necessità di sviluppo responsivo, Single Page Application (SPA) e Progressive Web Application (PWA)

2013: MEAN (MongoDB, Express, Angular e Node), soluzione full-stack:

- **Angular:** framework di scripting client-side JS
- **Node:** ambiente runtime server-side per eseguire JS
- **Express:** framework di scripting server-side
- **Mongo-DB:** il database No-Sql

NUOVE TECNOLOGIE

2006, Amazon Web Service (AWS) -> Cloud, “X come servizio”, risorse scalabili

2010, Container: come le VM ma condividono solo kernel dell’host e utilizzano solo le risorse di cui hanno bisogno- > **DOCKER, KUBERNETES, OPENSHIFT, RANCHER**

HTTP

Protocollo applicativo client-server, le specifiche sono gestite dal World Wide Web Consortium (W3C).

Porta (solitamente) 80: Hypertext Transfer Protocol, si basa su **http requests e repsonses.**

Questi messaggi sono suddivisi in 3 parti:

Start line: una riga di testo terminata da CRLF che descrive il MSG;

Header: molteplici righe terminate da CRLF che descrivono opziono e proprietà del messaggio. **Termina con una riga vuota.**

Body: dati generici facenti parti dell’oggetto del messaggio.

STRUTTURA MESSAGGI REQUEST:

1. **Start line:** <method><request-URL><version>
2. **Header:** <name1>:<value1> <name2>:<value2>…
3. **Body:** dati, anche in formato binario, o CRLF in caso non siano presenti

<method>

- **GET: chiede al server di inviare una certa risorsa al client;**
- **HEAD:** simile a GET, chiede però al server di inviare solo l’header e non il body della risorsa;
- **PUT:** chiede al server di memorizzare una certa risorsa inviata dal client (inverso di GET);
- **POST: utilizzato per inviare dati generici al server; a diffrenza di put, i dati non vengono memorizzati ma utilizzati da software esterni;**
- **TRACE:** utilizzato per ottenere informazioni diagnostiche sulla topologia di connessione;
- **OPTIONS:** chiede al server una lista delle funzionalità supportate;
- **DELETE:** chiede al server di rimuovere una determinata risorsa;

STRUTTURA MESSAGGI RESPONSE:

1. **Start line:** < version><status><reason-phrase>
2. **Header:** <name1>:<value1> <name2>:<value2>…
3. **Body:** dati, anche in formato binario, o CRLF in caso non siano presenti

<status>

- **Informational responses(100-199)**
- **Successful responses(200-299)**
- **Redirection messages(300-399)**
- **Client error reponses(400-499)**
- **Server error responses(500-599)**

STATUS PIÙ UTILIZZATI:

- **200: OK;**
- **301: moved permanently:** tutto trasferito ad altro URI (specificato in header location);
- **307: Temporany Redirect:** richiesta trasferita ad altra URI, ma in futuro si dovrebbe poter comunque raggiungere l'originale;
- **308: Permanent Redirect:** tutte le richieste verranno reinoltrate ad altro URI;
- **400 Bad Request:** la richiesta non può essere soddisfatta per errori di sintassi;
- **401 Unauthorized:** Autenticazione fallita o non fornibile;
- **403 Forbidden:** richiesta legittima ma il server non può soddisfarla; non ha attinenza con l'autenticazione (al contrario di 401);
- **404 Not Found:** la richiesta non è stata trovata;
- **405 Method not allowed:** la richiesta è stata effettuata con un metodo non permesso (i.e. usare GET per mandare dati che dovrebbero andare in POST);
- **408 Timeout:** il tempo per inviare la richiesta è scaduto e il server ha terminato la connessione;
- **500 Internal Server Error:** messaggio di errore generico;
- **502 Bad Gateway:** il server si comporta come un gateway o un proxy e ha ricevuto una richiesta invalida dal server di upstream;
- **503 Service Unavailable:** il server non è al momento disponibile;
- **504 Gateway Timeout:** il server si comporta come un gateway o un proxy e non ha ricevuto una risposta tempestiva dal server di upstream;

CAMPI IMPORTANTI IN HEADERS

GENERALI:

- **Date:** data e ora di generazione della richiesta;
- **Upgrade:** specifica una nuova versione del protocollo che il mittente desidera utilizzare;
- **Cache-control:** include direttive sul caching;
- **Trailer:** utilizzato per i messaggi "chunked", cioè spezzati in più parti;

REQUEST:

- **Client-IP:** ip del client;
- **Host IP:** ip del server;
- **Accept:** utilizzato per informare il server del tipo di file che il client supporta (i.e. JPEG, PNG...);
- **Accept-encoding:** utilizzato per informare il server del tipo di codifica che il client utilizza (i.e. Gzip);
- **If-Modified-Since:** richiede al server di restituire la risorsa solo se è stata modificata dopo il...
- **Authorization:** contiene i dati di auth del client (i.e. "username:password" in base64);
- **Cookie:** utilizzato per inviare generici token al server;

RESPONSE:

- **www-authenticate:**
- **Set-Cookie:**

- **Server:**
- **Allow:**
- **Content-Encoding:**
- **Content-Length:**
- **Content-Type:**
- **Last-Modified:**

LE VERSIONI DI HTTP

- **http/0.9 (1991):** supportava solo GET, no intestazioni o codici di stato;
- **http/1.0(1996):** supporta intestazioni e codici di stato;
- **http/1.1(1997):** connessioni persistenti, codice 304 (Not Modified), il metodo OPTIONS, i digest, decodifica i dati trasmessi con gzip;
- **http/2.0(2015):** supporto per richieste multiple in parallelo sulla stessa connessione e compressione degli headers, push di risorse da server a client;
- **http/3.0(2019):** QUIC(Quick UDP Internet Connection), protocollo di trasporto sviluppato da Google e basato su UDP;

http 2 molto utilizzato, ma http 1.1 più veloce quando si perdono pacchetti (utilizza una singola connessione TCP contro le 6 circa utilizzate da http 1.1);

RIASSUMENDO

http 1.1=Richieste in formato testo + TLS+TCP+IP
http 2.0=Richieste in formato binario + TLS+TCP+IP
http 3.0= Richieste in formato binario + QUIC(+TLS)+**UDP**+IP

CONNESSIONI TCP

Passa attraverso http, su livello “application” standard del modello ISO/OSI; all’interno dell’URL sono codificati **IP e porta**, necessari alla connessione.

- <ip sorgente> <porta sorgente>

<ip destinazione> <porta destinazione> Trattasi di connessione **object-oriented** con le seguenti caratteristiche:

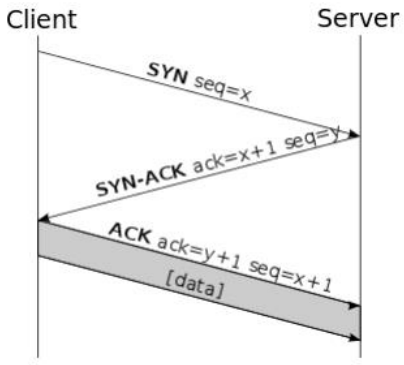
- **full-duplex:** può trasmettere e ricevere sulla stessa connessione;
- **point-to-point:** solo mittente e destinatario;
- **necessita dell’inizializzazione di variabili di stato da parte di mittente e destinatario;**

LE 3 FASI DELLA CONNESSIONE TCP

1. Set-up via **3-way handshake;**
2. Trasmissione dati;
3. Release della sessione;

Il 3-way handshake

- **A invia un segmento SYN a B:** il flag SYN è impostato a 1 e il campo **sequence number** contiene il valore X che specifica l'initial Sequence Number di A;
- **B invia un segmento SYN/ACK ad A:** i flag SYN ed ACK sono impostati ad 1, il sequence number contiene il valore y che specifica **l'Initial Sequence Number** di B ed il campo Acknowledgement number contiene il valore x+1 confermando la ricezione del ISN di A.
- **A invia un segmento ACK a B:** il flag ACK è impostato a 1 ed il campo Acknowledgment number contiene il valore y+1 confermando la ricezione del ISN di B.



Una volta stabilita la connessione, inizia la trasmissione dei dati tramite flusso: TCP li **divide** in segmenti di dimensione predefinita (ad esempio, 1460 Byte) e li **numera** per assicurarsi che vengano ricevuti in maniera corretta; esistono metodi per **controllare il flusso**, come **il controllo del buffer** e sistemi per la **conferma della ricezione** dei pacchetti tramite **ACK**.

In caso di perdita di pacchetti, è presente un meccanismo di **ritrasmissione, basato sulle conferme di ricezione.**

Quando terminano i dati da trasmettere, la connessione viene chiusa tramite **close (4-way handshake) tramite pacchetti FIN(ish) e ACK.**

ESEMPIO DI CONNESSIONE HTTP

Devo connettermi all'indirizzo: http://ww.prova.it:80/mypage.html

1) Il browser estrae l'hostname (www.prova.it);
2) Il browser cerca l'IP associato all'Hostname (tramite DNS): ad esempio, 1.2.3.4;
3) Il browser estrae la porta (:80);
4) Il browser effettua una connessione TCP verso l'ip associato tramite la porta (80:1.2.3.4)
5) Il browser effettua una http-request di tipo GET al server;
6) Il browser legge l'http-response del server;
7) Il browser chiude la connessione;
8) Una transazione http standard prevede:
    a. Di instaurare una connessione TCP;
    b. L'invio di un messaggio request (client-server);
    c. L'invio di un messaggio response (server-client);
    d. La chiusura della connessione;

le risorse http hanno un tipo , detto MIME (Multipurpose Internet Mail Extension), che descrive il loro formato. Si tratta di una semplice stringa di testo formattata con <tipo>/<sottotipo>, come:

- text/html: testo formattato HTML
- text/plain: testo con semplice codifica ASCII
- image/jpeg: immagine JPEG
- etc...

Ogni risorsa è identificata tramite **URI (Uniform Resource Identifier)**, che a sua volta è suddivisa in due tipologie: Uniform Resource Locator (**URL**) e Uniform Resource Name (**URN**). Il primo è l'indirizzo con il nome e identifica la risorsa in base alla sua posizione, il secondo è il nome proprio.

Schema di un URL

protocollo://host.dominio:porta/path?id=query=12(e parametri della query)

METHOD

GET: richieste da client a server; vengono inviate come parte dell'URI e sono facilmente intelligibili; le richieste sono sicure, non modificano dati sul server e possono essere facilmente memorizzate nella cronologia.

POST: invia informazioni al server per l'aggiornamento o la creazione di una risorsa; le informazioni vengono **inviate come parte del corpo della richiesta,** e quindi non direttamente intelligibili; sono meno sicure di GET e non possono essere memorizzate come URL dirette. Per contro, è meno limitata nella quantità di dati che scambia.

Esistono strumenti, come i dev-tools dei browser o add-on come POSTMAN che permettono di analizzare le richieste POST.

HTTPS

Strato di sicurezza intermedio tra http e tcp, noto come SSL (Secure Socket Layer) o TLS (Transport Layer Security).

HTTPS = HyperText Transfer Protocol over Secure Socket Layer

Fornisce:

- Autenticazione del server
- Autenticazione del client
- Integrità della trasmissione
- Cifratura della trasmissione

La cifratura è data da crittografia a chiave simmetrica, a chiave asimmetrica, firma digitale e certificati.

Nella firma digitale si utilizza un metodo di **cifratura asimmetrica a doppia chiave** (privata del mittente, decifrata dal destinatario con chiave pubblica del mittente per la non ripudiabilità, pubblica del destinatario decifrata con privata del destinatario per la confidenzialità)

Il certificato digitale nella crittografia asimmetrica è un certificato, emesso da CA, che attesta che la chiave pubblica in esso inclusa appartiene effettivamente ad una certa persona. Pertanto, esso sarà cifrato con la chiave privata della CA e decifrabile tramite apposita chiave pubblica.

SSL e TLS

- Il browser invia una richiesta di connessione al server;
- Il server invia il proprio certificato, contenente la sua chiave pubblica;
- Il browser verifica l'autenticità del certificato tramite C.A.; se è corretto, il browser crea una chiave simmetrica e la cifra con la chiave pubblica del server;
- Il server decifra la chiave simmetrica cifrata utilizzando la sua chiave privata;

Stateless e stateful

Un server senza stati è un server che tratta ogni richiesta come transazione **indipendente** non correlata ad ogni richiesta;

Un server con stati ha una tabella che permette di **registrare in modo incrementale le informazioni** e lo stato delle interazioni in corso con il client, in modo da poterle tracciare.

Esistono modi semplici basati sulle informazioni presenti negli headers per identificare i client, quali lo **user-agent**, il **referer** (la pagina da cui proviene il client tramite un link), il **client-ip.**

Purtroppo sono soluzioni deboli, in quanto nessuno dei campi è obbligatorio e la presenza di NAT e proxy potrebbe inficiare il sistema.

- **Forward-proxy:** raccoglie e instrada tutte le connessioni delle macchine di una LAN verso la rete esterna su un'unica interfaccia.
- **Reverse-proxy:** prende tutte le richieste provenienti dalla rete e tramite NAT le instrada sulle rispettive macchine di una LAN.

I **Proxy** possono fare **caching di contenuti statici e dinamici**, evitando di intasare di richieste su contenuti già consultati i server a monte e **possono comprimere i dati** per funzioni di balancing della rete.

Una delle strategie utilizzabili per tracciare le sessioni è quella di assegnare un ID univoco a tutte le url di una determinata web-app la prima volta che viene consultata da un client.

Cookies

I cookies sono stringhe di dati di piccole dimensioni che vengono memorizzate direttamente nel browser e che fanno parte del protocollo http.

Esse vengono impostate da un server tramite appositi comandi (**Set-Cookie http-Header**).

I coockie, tramite gli ID di sessione, sono in grado ad esempio di ricordarsi quali carrelli di un e-commerce stia gestendo un client in particolare.

L'id viene generato dal server ed inviato al client tutte le volte che esso visita una pagina e non ha precedenti cookie; in quel modo, il server potrà tenere traccia ad esempio di tutti gli URL visitati da quel client, degli articoli visionati e salvare questi dati in appositi registri.

Esistono diversi tipi di cookie:

- **Session cookie:** quando l'utente chiude il browser, vengono cancellati;
- **Persistent cookie:** persistono anche dopo la chiusura del browser;
- Secure cookie: viene trasmesso solo via HTTPS;
- **HttpOnly-cookie:** cookie che non possono essere letti da API client-side come Javascript; evita il furto di cookie tramite **cross-site scripting**;
- **SameSite-cookie:** sono cookie che possono essere inviati solo se il Server A ha anche fornito la risorsa (pagina html) contenente il link ad una risorsa contenuta in A. Evita il **cross-site request forgery;**

Assumono particolare rilievo gli attributi domain e path del cookie; nel primo caso, specificheremo un TLD e il cookie sarà valido sia su di esso che anche sui suoi sottodomini.

Nel caso di path, invece, il dominio sarà mutuato dalla richiesta http, e il cookie sarà valido solo nella path specificata di quel dominio.

È possibile impostare cookie relativi a domini del **sito corrente (first-part cookie)** o cookie relativi, ad esempio, a **contenuti esterni linkati nel nostro dominio (third-party cookies)** che tengano traccia degli spostamenti dell'utente tra i vari siti.

I cookie sono suscettibili ad alcuni problemi:

- **Network eavesdropping:** se non si usa HTTPS il canale è in chiaro e i cookie possono essere intercettati e letti.
- **Cross-site scripting:** senza l'attributo http-only le API JS possono accedere a tutti i cookies di un dominio.
- **Cross-site request forgery:** si utilizza l'autenticazione di un utente gestita tramite i cookie e la si usa per fargli svolgere azioni diverse.

I cookie identificano una combinazione di fattori (Browser, computer ed account). Se uno di essi cambia, i cookie perdono di efficacia.

Struttura dei cookie

Un cookie è composto da:

- Nome;
- valore;
- zero o più attributi, identificati come coppia nome/valore;

I browser non includono i valori nelle richieste al server, ma solo il nome ed il valore.

HTTP request header

Cookie: name1 = value1 [; name2=value2]...

HTTP response header

Set-cookie: name=value [; expires=date] [; max-age=numbre] [; path=path_value] [; domain=domain_value] [; scure] [; HttpOnly] [; Samesite]

CLIENT-SERVER

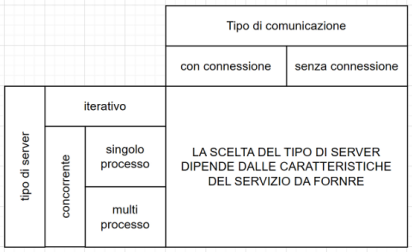
- **Client:** chi richiede il servizio;
- **Server:** chi offre il servizio;

Il server gestisce contemporaneamente più richieste da molti client. Esistono due tipi di iterazione:

- **connection oriented:** viene stabilita una connessione prima di scambiare contenuti;
- **connectionless:** no connessione, solo scambio messaggi;

Esistono anche diversi tipi di server:

- **iterativo:** processa le richieste di servizio una alla vota;
- **concorrente, a singolo o multiprocesso:** gestisce molte richieste contemporaneamente, la gestione di più processi contemporaneamente comporta una precisa sincronizzazione ed atomizzazione nell'accesso alle risorse;



API

Insieme di protocolli e servizi per la creazione e l'integrazione del software applicativo.

Esistono 3 tipi di relaese per le API:

- **Private:** API ad uso interno;
- **Partner:** API condivise con partner commerciali specifici;
- **Public:** API disponibili per tutti;

Remote API: APE pensate e realizzate per funzionare attraverso una rete di comunicazione; ad esempio le Web API, pensate per funzionare su http. In questo caso, le risposte vengono fornite tramite file JSON o XML.

I due approcci architetturali che utilizzano maggiormente le remote API sono:

- **Service Oriented Architecture**
- **Microservice Architecture**

Il primo è un approccio più monolitico e vecchio, il secondo è più agile e snello, implementa sistemi più piccoli e specializzati, che condividono tra loro framework di messaggistica comune (tipo RESTFul API) senza complicate conversioni di dati da un servizio all'altro.

Soa è quindi un paradigma generale per la creazione di servizi, i microservizi sono una implementazione di SOA che si concentra sulla realizzazione di servizi più piccoli ed indipendenti.

SOAP e REST

SOAP (Simple Object Access Protocol): è un protocollo, nato per la comunicazione tra reti SOA.

REST (REpresentaional State Tansfer): è uno **stile architetturale**; le API che lo seguono (denominate RESTful API) servono alla comunicazione tra **Microservices**.

I 6 vincoli del sistema RESTful:

1. **Architettura client-server**, via http;
2. **Server stateless**;
3. **Cacheability;**
4. **Layered system:** è possibile inserire livelli aggiuntivi alle comunicazioni client-server, quali load balancing e sicurezza;
5. **Code on demand:** i server possono estendere le funzionalità di un client trasferendo codice eseguibile;
6. **Uniform Interface:** 4 aspetti:
    a. Identificazione delle risorse nelle richieste;
    b. Manipolazione delle risorse tramite rappresentazioni;
    c. Messsaggi autodescrittivi;
    d. Hypermedia come motore del'applicazione;

Restful deve la sua popolarità (e delle API) ad altri modelli middle-ware di scambio dati, quali MOM (Message Oriented Middleware), un framework di comunicazione asincrona che gestisce l'interoperabilità attraverso lo scambio di messaggi. Da esso nascono due tipi di modelli di comunicazione asincrona:

- **Publish/subscribe:** meccanismo di distribuzione molti a molti con instradamento verso uno specifico canale con n client in ascolto (**broadcast**);
- **Point-to-point**: i messaggi prodotti vengono instradati presso una specifica entità;

Rispetto a questo modello, RESTful API ha molti vantaggi:

- **scalabilità:** il server non deve tenere traccia di alcuna informazione rispetto ai client;
- **Reliability:** in caso di disaster-recovery, non è necessario recuperare lo stato condiviso, basta l'applicazione stessa;
- **Più facile implementazione;**
- **Visibilità:** ciascuna richiesta è atomica e può essere monitorata facilmente;

XML (eXstensible Markup Language)

Markup language basato su SGML (Standard Generalized Markup Language); non è orientato solo alla visualizzazione (come HTML), ma un formato assolutamente generale per descrivere dati.

<nometag></nometag>

I tag devono essere **sempre** chiusi ed il linguaggio è **case-sensitive.**

Posso chiudere un tag appena aperto con il solo tag <tag/> al posto che con una coppia degli stessi;

La **DTD (Document Type Definition)** permette di creare nuovi linguaggi XML. Una sua versione più aggiornata, **la XSD (XML Schema Definition)** serve a definire la struttura di un documento XML.

Altre tecnologie legate ad XML

- **XLink:** serve a collegare due documenti XML tra loro, più avanzato dei semplici hyperlink html;
- **XSL (eXtsensible Stylesheet Language):** il documento con cui si descrive il foglio di stile di un documento XML.
- **XPath:** linguaggio per definire porzioni di un documento XML, alla base di altri strumenti XML come XQuery; implementa funzioni per la manipolazione di stringhe;
- **XPointer:** identifica univocamente precise porzioni di un documento XML;
- **XQuery:** linguaggio di query per documenti XML;
- **SVG (Scalable Vector Graphics):** standard per la creazione di elementi grafici vettoriali per il linguaggio XML;

Esempi di XML:

```
<artists>
      <artist>
            <name>Pablo</name>
            <surname>Picasso</surname>
            <date_of_birth>25/10/1881</date_of_birth>
      </artist>
      <artist>
            <name>Vasilij</name>
            <surname>Kandinskij</surname>
            <date_of_birth>04/12/1866</date_of_birth>
      </artist>
</artists>
```

<artist>: **Root node**

<artist>: first node/node element
I Vari <artist> sono **child element**

<surname>, <date_of_birth> sono **elements**

```
<person>
  <gender>female</gender>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

```
<person gender="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

Nel primo caso, <gender> è un **element**, nel secondo un **attribute**.

JSON

JSON (JavaScript Object Notation) è un formato semplice per lo scambio dei dati; è basato su due strutture:

un insieme di coppie nome/valore (record, struct, dizionario, elenco di chiavi...)

un elenco ordinato di valori (array, vettore, sequenza,,,)

Oggetto

- Serie **non ordinata** di coppie **nome/valore;**
- Inizia con una **{**;
- Finisce con una **}**;
- Ogni nome è seguito da **:**
- le coppie nome/valore sono separate da **,**

Es. {nome1:valore1, nome2:valore2, etc..}

Array

- Serie **ordinata** di valori;
- comincia con **[**;
- termina con **]**;
- i valori sono separati da **,**

Es. [valore1, valore2, valore3 etc...]

Value

Un value può essere una stringa tra virgolette, un numero, un valore booleano, un null, un oggetto o un array; queste strutture possono essere **annidate**.

Esempi:

- nome/valore: “name”:”Mario”
- lista nome/valore: {“name”:”Mario”, “surname”:”Rossi”}
- oggetto con una proprietà (nome): {“name”:”Mario”}
- array: “macchine”:[“Ferrari”, “Red Bull”, “McClaren”]
- oggetto con due proprietà (nome e cognome): {“name”:”Mario”, “cognome”:”Rossi”}

Esempio di traduzione da XML a JSON

```xml
<formula1>
    <teams>
        <team>
            <name>Ferrari</name>
            <nationality>Italy</nationality>
            <drivers>
                <driver>Leclers</driver>
                <driver>Sainz</driver>
            </drivers>
        </team>
        <team>
            <name>Red Bull</name>
            <nationality>Austria</nationality>
            <drivers>
                <driver>Verstappen</driver>
                <driver>Perez</driver>
            </drivers>
        </team>
    </teams>
</formula1>
```

JSON

```json
{
  "formula1": {
    "teams": {
      "team": [
        {
          "name": "Ferrari",
          "nationality": "Italy",
          "drivers": {
            "driver": [
              "Leclers",
              "Sainz"
            ]
          }
```

In Js, per parsare stringhe JSON:

JSON.parse()

In Js, per convertire un oggetto in JSON:

JSON.stringify()

Es. se ho questo JSON:

{“nome”:”Mario”, “cognome”:”Rossi”, “età”:”30”}

Posso usare parse per convertirlo in Js:

const obj = JSON.parse(‘{“nome”:”Mario”, “cognome”:”Rossi”, “età”:”30”}’);

HTML

Hypertext Markup Language: linguaggio a marcatori che definisce i nodi dell'ipetertesto.

Caratteristiche:

- **testo codificato** tramite **coded character set** (ASCII, ISO A8859/ANSI, Unicode 16/32 bit, **UTF-8**);
- **Tag/mark-up,** che rappresentano le caratteristiche del testo;
- **Gramamtica** che regola l'uso del Mark-up;
- **Semantica** che definisce il dominio di applicazione del mark-up;

Tipi di linguaggi di Mark-up:

1. **procedurali** (o **imperativi**): il markup indica quale operazioni un dato programma deve compiere su un documento per ottenere una determinata presentazione (Es. LaTex)
2. **dichiarativi** (o **descrittivi**): il mark-up descrive la struttura di un documento testuale, identificandone i componenti (SGML, HTML, XML);

SGML (Structured General Model Language): Standard ISO del 1986, è l'antenato di HTML.

HTML è una applicazione/semplificazione di SGML.

HTML 4.0: 1999, inserisce i CSS. Nel 2000 esce XHTML 1.0, che usa XML 2.0 al posto di SGML come metalinguaggio di Mark-up. W3C decide di riformulare HTML 4 e continuare solo su XHTML.

4 Giugno 2004: nasce il **WHAT** (Web Application Hypertext Technology).

La spinta è verso XHTML 2.0 che viene tuttavia poco considerato per la sua non-retrocompatibilità con XHTML 1.1.

26 ottobre 2006; Tim Berners Lee pubblica "Reinventing HTML", ammettendo gli errori commessi in XHTML 2.0.

2009: Tim Berners "abbandona" l'appoggio a XHTML per dedicarsi ad HTML 5.0;

W3C:

2014: HTML 5
2016: HTML 5.1
2017: HTML 5.2
2021: HTML 5.3

HTML 5: rinnovato supporto alle API JAVASCRIPT, CSS, SVG e MathML.

TOURING COMPLETO

Un sistema in grado di risolvere qualunque problema risolvibile da qualunque altro computer esistente o di cui si può immaginare l'esistenza.

Requisiti per la completezza di un sistema di Touring:

Ramificazione convenzionale: capacità di determinare se una cosa è vera o falsa in base allo stato di un'altra cosa;

Memoria arbitraria: la macchina può cadere vittima di un problema **indecidibile.**

Nessun sistema possiede una vera memoria arbitraria; semplicemente, di fronte ad un problema indecidibile la macchina si fermerà. Quindi il secondo criterio viene spesso ignorato.

CSS+HTML+user input is Turing complete.

HTML: linguaggio

Tag:

- <tag> - apertura
- </tag> - chiusura
- <tag /> - apertura e chiusura

In XHTML ogni tag aperto deve essere chiuso ed ogni parte della struttura deve essere dichiarata in modo esplicito e non ambiguo;

Ecco alcuni esempi di elementi vuoti:

<input type=”hidden” name=”nome” value=”test” />
lo spazio prima dello / permette la retrocompatibilità con versioni di browser precedenti ad XHTML.

Gli **attributi** descrivono le proprietà degli elementi (nomeattributo=valore), es:

<elemento1 attributo1=valore1 attributo2=valore2>

Contenuto

</elemento1>

L’ordine degli attributi non è importante.

Differenze tra HTML e XHTML per l’uso degli attributi e dei tag:

- **in XHTML, il valore di un attributo deve essere inserito tra doppi apici** (attrib1=”valore1”), in HTML non è necessario;
- **in XHTML il nome di elementi ed attributi deve essere scritto in minuscolo,** mentre in HTML fino al 4.01 potevano essere anche maiuscoli;

In HTML gli elementi possono essere innestati uno dentro l’altro per creare una struttura gerarchica, a patto di chiudere sempre l’ultimo tag aperto prima di chiuderne altri

<tag1>

<tag2> elemento
</tag2>

</tag1>

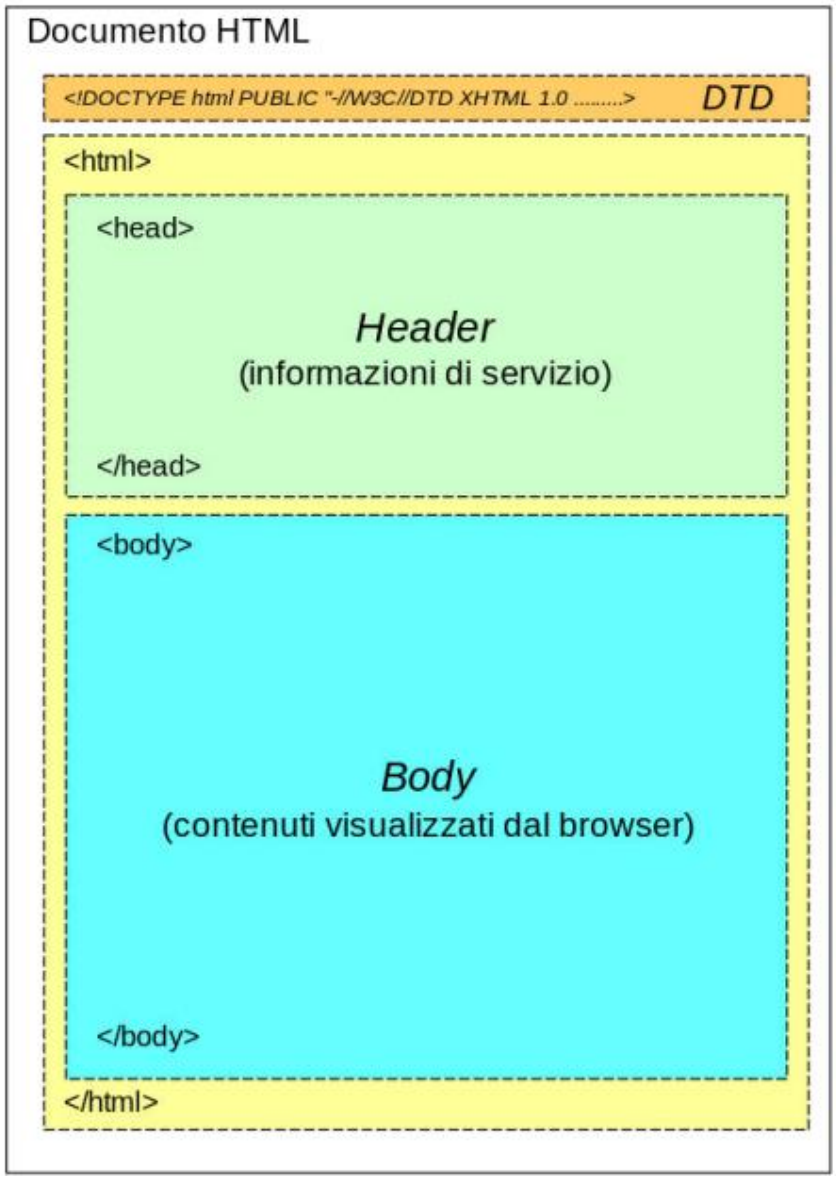
CODIFICA

1 byte= 8 bit = 256 caratteri

Codifica ASCII: 7 bit (128 caratteri): a-z A-Z, 0-9, ,.:-+/, caratteri di controllo e simboli (tab, a capo...)

- In Xml si usa **Unicode con UTF-8;** esso deve essere esplicitato nell'intestazione del documento XHTML o XML, mgari utilizzando l'intestazione content-type del protocollo HTTP:
  Content-type: text/html; charset=utf-8
- Per i documenti XML si usa invece lo pseudo-attributo encoding nella dichiarazione xml a inizio documento:
  <?xml version="1.0" encoding="utf-8" ?>
- Per i documenti HTML si usa l'elemento meta all'interno del tag HEAD:
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
- Per i documenti HTML5 sarà sufficiente all'interno dei tag head:
  <meta charset="utf-8/">

Struttura di base di un documento HTML



DOM (Document Object Model)

Suddiviso in 3 parti:

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

HTML DOM: standard per doc HTML L'HTML DOM definisce:

- Gli **elementi** HTML come oggetti
- Le **proprietà** di tutti gli elementi HTML
- I **metodi** per accedere a tutti gli elementi HTML
- Gli **eventi** per tutti gli elementi HTML

La pagina risulta costituita da **nodi**: ogni tag, ogni testo è un nodo. I nodi possono avere attributi (come a href="...") e **proprietà**. Ogni nodo può contenere altri nodi; in questo caso si parla di **elementi**.

Un **elemento** è contraddistinto da un **tag**, qualunque esso sia.

In **nodo** ha un significato semantico più ampio; può anche essere un testo, un attributo o altro. Un elemento è il tipo specifico di un nodo, **quindi un elemento eredita da un nodo.d**

Il **document** è l'elemento che contiene tutti gli altri elementi del DOM.

INTESTAZIONE

HTML= DTD (Document Type Definition)

es. <!DOCYTPE HTML PUBLIC="-//W3C//DTD HTML 4.01 Transitional//EN" http://www.w3c.org/TR/html4/loose.dtd> o in HTML 5 <!DOCTYPE html>

Analisi:

- HTML: linguaggio;
- PUBLIC: il documento è pubblico;
- W3C: ente che ha rilasciato le specifiche;
- DTD HTML 4.01 Transitional: versione del linguaggio;
- EN: lingua in cui è scritto il DTD;
- http://... : specifiche del linguaggio

HEADER

<head>...</head>

Attributi:

- **<title>:** titolo pagina;
- **<meta>:** metadati da passare ai motori di ricerca;
- **<base>:** come vengono gestiti i riferimenti relativi nei link (specifica la URI assoluta dei link relativi presenti nel documento);
- **<link>:** collegamenti verso file esterni (i.e. CSS);
- **<script>:** codice eseguibile utilizzato dal documento;
- **<style>:** informazioni di stile;

Su **link**, gli attributi possibili sono:

- **href:** specifica la URI del documento correlato;
- **type:** contiene il 'content type' del documento correlato in formato MIME, i.e.: text/css, text/plain;
- **rel, rev:** specificano la relazione tra il doc corrente ed il correlato (i.e. next, prev, stylesheet);
- **Stylesheet:** indica foglio di stile esterno, spesso utilizzato insieme al link di tipo Alternate che permette di selezionare un fogli di stile alternativo a quello utilizzato;
- Next: in una sequenza di doc, indica quello successivo, utile per precaricare i doc e ridurre i tempi;
- **Prev:** documento precedente;
- **Start:** indica il primo doc di una serie, utile per fornire metadati ai motori in presenza di una serie di doc;
- **Alternate:** una versione sostitutiva di un doc; spesso usato insieme a lang per fornire alternative di una risorsa in varie lingue;

Su **script**, gli attributi possibili sono:

- **language:** linguaggio di scripting;
- **type:** contenuto dello script in formato MIME;
- **src:** URI contenente il codice di scripting da eseguire;

Su **meta**, gli attributi possibili sono:

- **http-equiv:** informazioni al browser su come gestire la pagina (i.e. <meta http-equiv=http-equiv-name content=valore>). I valori possibili per http-equiv-name sono:
  - **refresh:** indica che la pagina deve essere ricaricata dopo tot secondi;
  - **expires:** stabilisce la data di scadenza per il documento;
  - **content-type:** indica il tipo di contenuto in MIME (i.e. <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">)
- **name:** fornisce informazioni utili ma non critiche con la seguente struttura <meta name=name-value content=valore>. Ecco alcuni possibili valori di name-value:
  - **author;**
  - **description;**
  - **copyright;**
  - **keywords:** parole chiave per i motori di ricerca separate da una virgola (i.e. <meta http-equiv=keywords lang="en" content="chiave1, chiave2, chiave3");
  - **data:** la data di creazione del documento;

BODY

<body>...</body>

Alcuni attributi possibili:

- **background=url** immagine di sfondo da usare per la pagina;
- **text=color** colore del testo;
- **bgcolor=color** colore di sfondo;
- **lang=linguaggio**;

Tipi di elementi includibili in body:

- **Intestazioni:** titoli in gerarchia;
- **Strutture di testo:** paragrafi, testo indentato…
- **Aspetto del testo:** grassivo, corsivo etc…
- **Elenchi e liste:** a numeri, puntati…
- **Tabelle;**
- **Form;**
- **Collegamenti ipertestuali ed ancore;**
- **Immagini e contenuti multimediali;**
- **Contenuti interattivi** (script, app esterne)

3 tipi di elementi di layout:

1. **Block-level**: fanno blocco, quindi vanno a capo (paragrafi, tabelle, form…)
2. **Inline:** non vanno a capo e si integrano nel testo (immagini, link…)
3. **Liste:** numerate, puntate…

ELEMENTI REPLACED: hanno caratteristiche intrinseche definite (altezza, larghezza etc..) dall'elemento stesso e non da quello che lo circonda.

ELEMENTI NON REPLACED:

- **Headings (<h1>, <h2>…):** definiscono la grandezza di titoli di importanza crescente (da 1 a 6); ammettono attributi di allineamento (left | center | right | justify);
- **Paragrafi (<p>): lascia una riga vuota prima dell'apertura e dopo la chiusura; per andare a capo all'interno del paragrafo si usa
;**
- **Span ():** contenitore generico annidato all'interno dei ; **non va a capo (inline).**
- **Text container (<p>, ,): div non lascia spazio prima o dopo ma va a capo.**

Stili del testo

Tag fisici: definiscono lo stile del carattere in termini grafici **indipendentemente** dalla funzione del testo nel documento:

- **:** grassetto;
- **<code>:** monospaziato;
- **<blockquote>:** citazione, rientro a destra;

Tag logici: forniscono informazioni sul ruolo svolto dal contenuto e, in base ad esso, adottano uno stile grafico:

- **<tt>:** monospaziato;
- **<i>:** corsivo;
- **<b>:** grassetto;
- **<u>:** sottolineato;
- **<s>:** testo barrato;

Liste:

- ** (unordered list)** per le liste puntate; ogni elemento viene identificato con il tag ** (list item)**;
- ** (ordered list)** per liste numerate; ogni elemento viene identificato con il tag ** (list item);**

TABLE:

il tag <table> racchiude una serie di attributi:

- **align (left | center | right);**
- **width:** larghezza, o numerica in px o percentuale rispetto alla pagina;
- **bgcolor="#xxxxxx";**
- **border="n"** spessore bordi;
- **cellspacing, cellpadding;**

RIGHE DELLA TABELLA (<tr>, table row):

- **align (left | center | right);**
- **valign(top | middle | bottom | baseline)** allineamento verticale;
- **bgcolor="#xxxxxx";**

TH e TD (CELLE)

- **<th>:** celle testata;
- **<td>:** celle righe;

Le celle hanno gli stessi attributi possibili per le righe più:

- **width, height** = {length | length%} dimensioni della singola cella;
- **rowspan, colspan** = n indica su quante righe/colonne della tabella si estende la singola cella;

LINK IPERTESTUALI

Possono avere l'attributo fragment (#xxxxxx) per indirizzare ad un punto specifico della risorsa linkata;

IMMAGINI

<img src="image_name.ext">

Attributi:

- **src**= url dell'immagine;
- **alt** = testo alternativo nel caso fosse impossibile visualizzare l'immagine;
- **width, height;**

HTML 5

Inserisce l'elemento <canvas> che permette di disegnare direttamente sulla pagina ed interagire tramite JS con elementi mutimediali avanzati, senza l'utilizzo di plugin esterni i.e per la visualizzazione di contenuti.

Inserisce l'elemento <video> e <audio> (spesso però dovendo codificare con più codec i contenuti, visto che non esiste un codec universale).

Inserisce inoltre API specifiche per i servizi di geolocalizzazione, con metodi quali:

- **getCurrentPosition():** restituisce lat e long della posizione dell'utente;
- **watchPosition():** restituisce la posizione corrente e continua ad aggiornare la posizione dello stesso durante i suoi spostamenti;
- **ClearWatch():** termina il metodo watchPosition();

AUTH in http

- **Basic access Authentication**
    a. Client fa http request:
    GET /www.mybank.it/credicardcodes.html HTTP/1.1
    b. Server risponde 401 (non autorizzato) con un messaggio che esplicita:
    HTTP/1.1 401 Unauthorized
    …
    WWW-Authenticate: Basic realm="InternetSecurity"
    …
    Corpo messaggio di risposta
    c. Il client invia nuova req esplicitando user e pass:
    GET /www.mybank.it/credicardcodes.html HTTP/1.1
    …
    Authorization Basic: QWxhZGRpbjpvcGVuIHNlc2FtZQ==
    …

    La stringa è in radx-64 concatenando i sue valori (user:pass).

Questo tipo di accesso esprime tuttavia in chiaro le credenziali.

- **Digest Access Authentication**
    Funziona tutto come prima ma, nel messaggio di risposa del server sono esplicitati i seguenti campi:

    …
    WWW-Authenticate: Digest
    Realm="testrealm@host.com",
    Qop="auth-int",
    nonce="fkdls2342lfdiornN6345LnSFE",
    opaque="mngakdsn43q329429sfnmnv20KAGAF",
    …
    Corpo del messaggio di risposta
    Il client risponde alla sfida esplicitando i valori con questa logica:
    GET /www.mybank.it/creditcardcodes.html HTTP/1.1
    …
    WWW-Authenticate: Digest
    Username="myuername",

Realm=”testrealm@host.com”,
Qop=”auth-int”,
nonce=”fkdls2342lfdiornN6345LnSFE”,
opaque=”mngakdsn43q329429sfnmnv20KAGAF”,
url=” www.mybank.it/creditcardcodes.html”,
nc=000000001,
cnonce=”0a4f113b”
response=”afjkl335nk4nndsfkfkljsd1092NSnma”,
…
Essendo:
cnonce: valore casuale e/o meccanismo di mutua autenticazione;
response: riporta il digest MD5 prodotto da client http generato con la seguente logica:
H(H(A1) : nonce : nc : cnonce : qop : H(A2))
Con:
A1= username : realm : user – password
A2= method : url : H(body)

Altri metodi di autenticazione

- **Auth con public key:** utilizza un **certificato** del client (HTTPS/SSL);
- **Kerberos o SPNEGO:** utilizzati da Microsoft IIS;
- **Secure remote password;**
- **OAuth2:** utilizza JSON Web Token (JWT) e Bearer Token;

OAuth2 è il più utilizzato; si tratta di una serie di valori JSON serializzati. È suddiviso in 3 parti, codificate in base64:

- **Header:** contiene il tipo di contenuto del payload e il tipo di algoritmo usato per il signature (RS256 asimmetrico con chiave privata per la generazione e pubblica per la validazione oppure HS256 simmetrico);
- **Payload**: contiene i claim che vogliono trasmettere le parti;
- **Signature**

I Bearer Token sono una stringa hex opaca che autorizzano l’accesso ad una singola risorsa protetta da una Auth server dedicato; se si entra in possesso di un bearer Token, si può accedere alla risorsa.

FORMS

<form action=mailto:xxx@yyy.zz method=”post”

Enctype=”text/plain”

<input type=”text” name=”subject” value=”ciao”/>
<input type=”submit” name=”empty” value=”invia” />
L’attributo **action** è basato su un valore URI che determina l’azione del form stesso (http, mail).

L’attributo **method** indica il metodo http che sarà usato al momento dell’invio dei dati; in caso di post, esso è suddiviso in due parti:

1. il client comunica al server che invierà dei dati in post all'indirizzo in action;
2. in una trasmissione apposita, il client trasmette al server di dati del form;

In caso di **get**, i dati sono inviati in un unico passaggio.

L'attributo **enctype** indica il formato di codifica usato da form per trasmettere i dati; può essere:

- **application/x-www-form-urlencoded:** valore predefinito, converte tutti i valori del campo form in coppie nomecampo=valore;
- **test/plain:** usato dalle form che hanno action mailto;
- **multipart/form-data:** si usa solo con method="post"; codifica ogni campo del form in una o più parti di un documento **MIME (Multipurpose Internet Mail Extensions – RFC 2045-2049)**;

L'attributo **name** specifica il nome della form.

L'attributo **target** specifica dove mostrare la risposta del server:

- **__blank**
- **__self**
- **__parent**
- **__top**

Form input

Attributi comuni:

- **name**
- **value**

type Type può assumere i seguenti valori:

- **text:** casella di testo monoriga;
- **password:** come text, ma sostituito da asterischi;
- **file:** permette di caricare un file;
- **checkbox:** casella di spunta;
- **radio:** radio button;
- **submit:** botton per inviare il contenuto del form;
- **image:** bottone di submit sotto forma di immagine;
- **reset:** riporta i campi al valore iniziale;
- **button:** bottone di azione;
- **hidden:** campo nascosto;

Esiste l'attributo **disabled** che permette di disabilitare gli input.

I bottoni hanno tre valori possibili per il campo **type**:

- **submit:** bottone che manda i dati del form al server;
- **reset:** bottone che riporta al valore inziale i campi;
- **button:** un generico bottone di azione;

Button permette di creare bottoni complessi.

CSS

Cascading Style Sheet: servono a separare l'HTML, che definisce testo e contenuto della pagina, da come deve essere presentato all'utente.

1993: Netscape e IE presentano tag proprietari per definire caratteristiche grafiche e stilistiche delle pagine web; erano spesso lunghi, mancavano di logica e non erano, ovviamente, responsive.

1990-2000: Adobe Flash

1996 - CSS 1: linguaggio di formattazione visiva

1998 – CSS 2: introduce il supporto per media multipli e linguaggio di layout sofisticato e complesso;

2004 – CSS 2.1: versione migliorata di CSS 2;

2009 – CSS3: è diviso in 4 moduli: Media Queries, Namespaces, Selectors, Color. Presenta nuovi selettori, pseudo-classi e pseudo-elementi. Supporto completo ad XML, si possono selezionare elementi specifici in base ad attributi o stringhe. Utilizzo condizionale su vari dispositivi di alcuni stili.

Cascading: la struttura ad albero permette di ereditare caratteristiche dal nodo padre, ma esse possono essere sovrascritte da caratteristiche specifiche.

Stylesheet: insieme di regole.

Regola: istruzione che permette di specificare l'aspetto stilistico di uno o più elementi.

Una regola è composta da 2 parti:

1. **Selettore:** parte prima delle parentesi graffe, specifica quali elementi sono assoggettati ad una regola, è il collegamento tra il documento HTML e lo stile.
2. **Dichiarazione:** parte dentro le graffe, è la parte di regola che imposta il comportamento stilistico degli oggetti coinvolti.

La **dichiarazione**, a sua volta, è composta da 2 parti, separate da **":"**

- **Proprietà:** la parte prima dei due punti, descrive una caratteristica che un oggetto può avere.
- **Valore:** la parte dopo i due punti, specifica quale comportamento deve avere la proprietà selezionata.

Le **regole** possono essere **raggruppate**:

h1, h2, h3 { color: green; font-weight: bold; text-align: center; }

Regole con lo stesso **selettore** possono essere contratte in una regola sola:

- h1 { color: green }
- h1 { font-weight: bold }
- h1 { text-align: center }

h1 { color: green; font-weight: bold; text-align: center; }

Associazione degli Stylesheet

- Applicando un **attributo style** ad un **elemento**;

i.e. ...
utile per elementi singoli con stile diverso dal solito o per test

- applicando **l'elemento sytle** ad un **documento**;
  <head>
  <style>
  Body {background-color: green;}
  H1 {color: red; margin-left: 10 px;}
  </style>
  </head>

- collegando uno **stylesheet esterno tramite link**;
  <head>
  <link rel="stylesheet" type="text/css" href="mys.css">
  </head>

Il ink esterno è **riusabile**, **conveniente a livello di prestazioni** (cacheato la prima volta), **selezionabile dall'utente** dove ne vengono offerti una serie.

In caso di compresenza più fogli di stile, si crea una nuova "cascata" con il seguente ordine di priorità:

1. Stile in linea (dentro un elemento HTML);
2. Fogli di stile interni ed esterni (in <head>);
3. Predefinito nel browser;

Il **selettore** serve per collegare uno stile agli elementi a cui deve essere applicato.

- **Selettore universale:** identifica qualunque elemento -> *** {...}**
- **Selettori di tipo:** si applicano a tutti gli elementi di un determinato tipo -> **tipo_elemento {...}**
- **Classi:** si applicano a tutti gli elementi che presentano l'attributo class="nome_classe" -> **.nome_classe {...}**
- **Identificatori:** si applicano agli elementi che presentano l'attributo id="nome_id" -> **#nome_id {...}**

I selettori di tipo si possono combinare con quelli di classe e di identificatore:

- tipo_elemento.nome_classe {...}
- tipo_elemento#nome_id {...}

Pseudoclassi: si applicano ad un sottoinsieme degli elementi di un tipo identificato da una proprietà **tipo_elemento:proprietà {...}** *es. l'hover di un elemento h1:hover {...}*

Pseudoelementi: si applicano ad una parte di un elemento **tipo_elemento:parte {...}** *es. la prima linea di un paragrafo p:first-line{...}*

Proprietà singole permettono di definire un singolo aspetto di stile;

Shorthand properties *definiscono un insieme di aspetti correlati fra di loro usando una sola proprietà; i.e. i margini p {margin-top: 10px; margin-right: 10px;...} -> p {margin: 10px 10px 10px 10px;}*

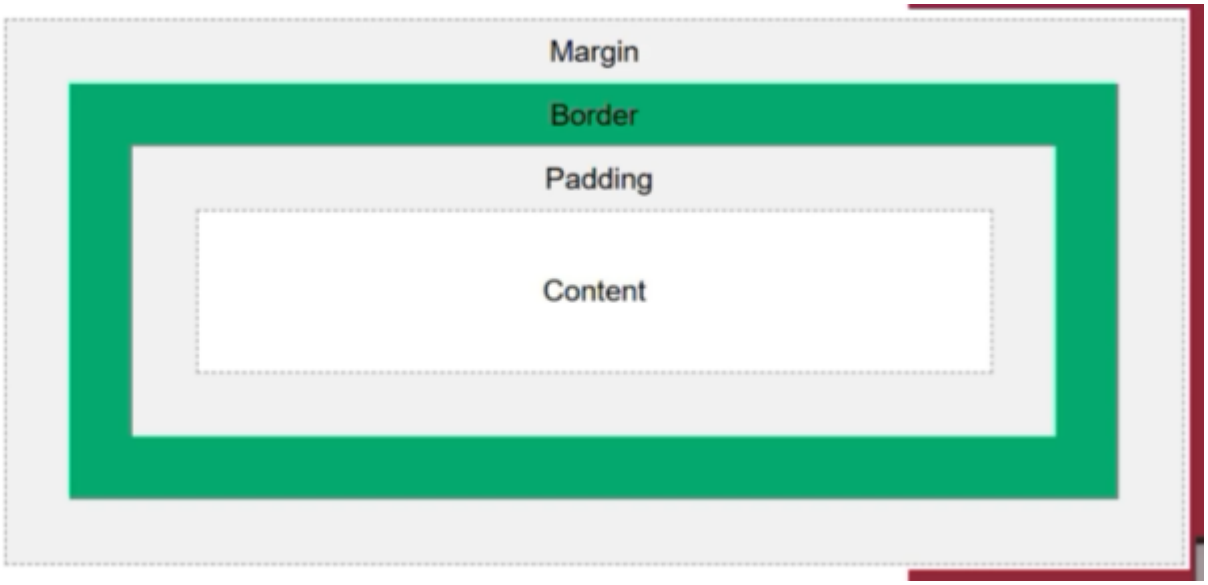
In questo caso l'ordine degli spazi è top-right-down-left

Regole di risoluzione sulla priorità in cascade

- **Origine**: l'ordine di prevalenza è autore, utente, browser;
- **Specificità del selettore:** ha precedenza il selettore con specificità maggiore;
- **Ordine di dichiarazione:** se esistono due dichiarazioni con egual specificità e origine vince quella fornita per ultima;

una regola marcata come **!important** ha sempre precedenza sulle altre.

CSS Box Model



- **Content**: il contenuto del box, dove possono comparire immagini e testo;
- **Padding**: spazio vuoto tra contenuto e bordo;
- **Border**: bordo che circonda padding e contenuto;
- **Margin** spazio vuoto tra l'elemento e quelli adiacenti;

OVERFLOW: comportamento da adottare quando il contenuto "deborda"; può essere "**Visible", "hidden", scroll e auto.**

Posizionamento e display

Display: proprietà che determina **come** un elemento viene visualizzato sulla pagina; può essere **inline, block , inline-block e none**;

Posizionamento: determina la posizione di un elemento rispetto ai suoi fratelli o rispetto al suo contenitore (**static, relative, absolute, fixed, float**);

In **CSS3** sono stati introdotti flex e grid:

- **flex**: l'elemento diventa un contenitore flessibile, in grado di allineare e distribuire gli elementi figli;
- **grid**: l'elemento diventa un contenitore a griglia, in grado di organizzare gli elementi figli su una griglia.

Position – legacy:

- **static**: posizionamento naturale del flusso;
- **absolute**: il box dell'elemento viene rimosso dal flusso ed è posizionato rispetto al box contenitore del primo elemento antenato "posizionato", ovvero non static;
- **relative**: l'elemento viene posizionato relativamente al box che l'elemento avrebbe occupato nel normale flusso del documento;
- **fixed**: posizionamento rispetto al viewport (browser window);

CAPITOLO 35 RIFARE TEST e ricercare anche fuori position: relative etc...

Javascript

Due tipi di linguaggi: interpretati e compilati. JS è **interpretato**.
Il codice JS deve essere inserito tramite i tag **<script></script>;**

```
<html>
<head>

    .....

    <!—POSIZIONE 1 -->

</head>
```

```
<body>
    <!—POSIZIONE 2 -->
    <p id=”myP”>Hello world</p>
    <!—POSIZIONE 3 -->
</body>
</html>
```

Nella posizione <head>, se lo script prevede l'utilizzo di tag in <body>, **potrebbe non funzionare correttamente.**

Idem se inserito in punti del body dove i tag necessari non sono ancora stati invocati.

È utile scrive il codice Js in file dedicato:

<script src=”script.js”></script>

In sintassi JS ci sono due tipi di valori:

- fissi
- variabili

Per dichiarare la variabili posso usare sia **var** che **let**; var ha un ambito di funzione (function scope), let ambito di blocco (block scope). **Inoltre, le variabili definite in let non possono essere ridefinite e devono essere dichiarate prima dell'utilizzo.**

JS gestisce 8 tipi di dato:

1. **String**
2. **Number: double (floating point a 64 bit)**
3. **Bigint: numeri interi più grandi di “Number”**
4. **Boolean**
5. **Undefined: variabile senza valore**
6. **Null**
7. **Symbol**
8. **Object**

Strutture condizionali utilizzate:

- **if**
- **else**
- **else if**

switch Cicli:

- **for**
- **for/in:** cicla attraverso le proprietà di un oggetto;
- **for/of:** cicla attraverso i valori di un oggetto iterabile
  Es.
  for (i=0; i<numeri.length; i++) {
  console.log(numeri[i]);
  }

- **while:** cicla attraverso un blocco di codice finché una specifica condizione è vera
- **do/while:** cicla anche attraverso un blocco di codice finché una specifica condizione è vera

Objects

È un insieme di proprietà (variabili, funzioni, altri oggetti) che possono essere accedute e manipolate; gli oggetti JS sono **dinamici**, quindi le loro proprietà possono essere aggiunte o rimosse dal codice in runtime.

Esempio sintassi:

```
const car = {
    make: 'Honda',
    model: 'Civic',
    year: 2020,
    isOn: false,
    turnOn: function() {
        thi.siOn = true;
        console.log('The car is on');
                }
     turnOff: function(){
    this.isOff = false;
    console.log('The car is off');
                }
};
```

Il loop for-in viene utilizzato proprio per iterare le proprietà di un oggetto:

for (var nomeProprietà in nome oggetto) {codice da eseguire}:

Es. di esecuzione di una funzione:

```
const persona = {
    nome: "Mario",
    saluta: function() {
        console.log("Ciao, sono " + this.nome)
        }
};


//output: "Ciao, mi chiamo mario"
persona.saluta();
```

Array

Permette di visualizzare una serie di valori (numeri, stringhe etc…) in una singola variabile.
Var nome Array = [valore1, valore2, valore3…]
Es. di uso in object
Var frutti = [‘Mela’, ‘Banana’, ‘Arancia’]
Frutti.forEach(function(frutto)) {

Console.log(frutto);

});

uso di push / pop:

var numeri = [1, 2, 3];
numeri.push(4);
console.log(numeri)

Var frutti = [‘Mela’, ‘Banana’, ‘Arancia’]
Var ultimoFrutto = frutti.pop();
console.log(frutti);
//output: [‘mela’, ‘banana’]

È possibile usare **splice** per aggiungere o rimuovere un elemento da una funzione specifica di un array.

Nomearray.splice [start, deleteCount, item1, item2, …]

- Start: indice da cui iniziare ad aggiungere o rimuovere elementi;
- deleteCount: il numero di elementi da rimuovere a partire dall’indice start;
- item1, item2 etc…: gli elementi da aggiungere all’array partendo dall’indice start;

TEST CAPITOLO 40 DA RIVEDERE, LE RISPOSTE SI BASANO SU CODICE CHE NON SI VEDE NELLE DOMANDE

Funzioni

Sintassi di definizione:

function nomeFunzione (parametro1, parametro2 etc…) {
//blocco di codice da eseguire

}

Sintassi di esecuzione:

nomeFunzione (argomento1, argomento2…);

- **Funzioni anonime (senza un nome), Es:**

  const quadrato = function(x) {
  return x * x;

  };

- **Funzioni assegnate ad una variabile:**

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

};
```

- **In alternativa, si possono usare le arrow function:**

```
(parametro 1, parametro 2, ...) => {
//blocco di codice da eseguire

}
```

Le arrow function **non presentano un valore di *this***, esso è preso dal contesto in cui la funzione è definita.

Non hanno inoltre *arguments:* le arrow function non dispongono di argomenti, ma vi accedono attraverso la sintassi.

Risultano, infine, più concise e leggibili di quelle tradizionali.

Funzione tradizionale	Arrow func corrispondente
function quadrato(x) { *return x * x;* *}*	const quadrato = (x) => x * x;

JS and Json

- **JSON.stringify():** da oggetto a JSON
- **JSON.parse():** da **stringa** JSON a oggetto

JS HTML DOM

Il DOM (Document Object Model) è un'interfaccia di programmazione delle applicazioni (API) che consente di manipolare gli elementi HTML e XML di una pagina web attraverso JS; è l'albero di una struttura di una pagina web ed ogni elemento è rappresentato da un oggetto del DOM.

Tramite esso è possibile accedere, creare e modificare gli elementi di una pagina web dinamicamente, senza ricaricare.

Metodi:

- **getElementById():** restituisce **l'elemento HTML che ha un attributo "id"** corrispondente al valore specificato come argomento:
  var elemento = document.getElementById("idElemento");
- **getElementsByClassName():** restituisce **una collezione di tutti gli elementi HTML che hanno una classe CSS corrispondente** al valore specificato come argomento:
  var elementi = document.getElementsByClassName("classElem");

- **getElementsByTagName():** restituisce **tutti gli elementi che hanno un tag HTML** corrispondente:
  var elementi = document.getElementsByTagName("Tag");
- **querySelector():** restituisce **il primo elemento HTML che corrisponde al selettore CSS** specificato come argomento:
  var element = document.querySelector("#idElement.classElement");
- **querySelectorAll(): restituisce una collezione di tutti gli elementi HTML che corrispondono al selettore CSS** specificato come argomento:
  var elementi = document.querySelectorAll(".classElement");

Selettori:

- **Selettore di tipo:** seleziona tutti gli elementi di tipo o <ul> in una pagina HTML;
- **Selettore di classe**: seleziona tutti gli elementi appartenenti alla classe "class" tramite la sintassi "**.class**";
- **Selettore di ID**: seleziona un singolo elemento in base all'ID, con la sintassi "#ID", ad esempio "**#header**";
- **Selettore discendente:** seleziona tutti gli elementi figli di un altro elemento; si indicano gli argomenti separati da uno spazio (se indico "**ul li**" selezionerà tutti gli elementi li che sono figli dell'elemento ul);
- **Selettore di figlio diretto:** seleziona solo gli elementi che sono figli diretti di un altro elemento; la sintassi è di separarli con ">" spazio (se indico "**ul > li**" selezionerà gli elementi li che sono figli diretti dell'elemento ul);
- **Selettore di attributo**: seleziona gli elementi tramite un attributo HTML specifico; utilizzo la sintassi **"[href]**", ad esempio, per selezionare tutti glie elementi con attributo "href";
- **Selettore di attributo con valore**: seleziona gli elementi tramite un attributo HTML specifico con determinato valore; utilizzo la sintassi **"[href='#']",** ad esempio, per selezionare tutti glie elementi con attributo "href" e valore settato a '#';
- **Selettore di pseudo-classe:** seleziona un elemento in base ad uno stato o una condizione specifica, ad esempio il rollover del mouse o un link già visitato; la sintassi è "**:hover**";
- **Selettore di pseudo-elemento:** seleziona parti specifiche di un elemento HTML quali, ad esempio, la prima lettere di un paragrafo; la sintassi è **"::first-letter".**

Proprietà:

è utile conoscerle per poterne manipolare il contenuto:

- **innerHTML**: restituisce il contenuto di un elemento HTML, comprensivo di eventuali tag HTML al suo interno; esempio di manipolazione:
  var elemento = document.getElementById("idElem");
  elemento.innerHTML = " new HTML content "
- **textContent:** restituisce il contenuto di un elemento HTML **senza** eventuali tag HTML al suo interno; esempio di manipolazione:
  var elemento = document.getElementById("idElem");
  elemento.innerHTML = "new text content"

- **value:** restituisce o imposta il valore di un elemento **input**, come ad esempio una casella di testo o una di selezione; viene utilizzata principalmente per gli elementi input e textarea. Esempio di manipolazione:

  var input = document.GetElementById("idInput");
  var valoreInput = input.value;
  input.value = "Nuovo valore";

Possiamo poi creare nuovi elementi HTML tramite il metodo **createElement;** questo metodo crea un nuovo elemento HTML e lo aggiunge alla pagina.

var nuovoElemento = document.createElement("div");
Il parametro specifica che tipo di elemento HTML creare; esso può poi essere personalizzato tramite le proprietà e i metodi DOM:
nuovoElemento.textContent = "Nuovo elemento";
nuovoElemento.style.color = "red";

Sarà poi necessario aggiungerlo alla pagina web tramite il metodo **appendChild** che aggiunge l'elemento alla fine dell'elemento padre specificato:

var elementoPadre = document.getElementById("idElemPadre");

elementoPadre.appendChild(nuovo elemento);

Posso anche inserire il nuovo elemento in posizione specifica tramite il metodo **insertBefore**, che inserisce il nuovo elemento prima di un elemento figlio specifico**:**

var elementoFiglio = document.getElementById("IdElemFiglio");

elementoPadre.insertBefore(nuovoElemento, elementoFiglio);

Posso anche rimuovere l'elemento tramite **removeChild**.

Var elem=document.getElementById("idElemDaRimuovere");

```
var elemPadre = elem.parentNode;
elemPadre.remove(elem);
```

Eventi:

rappresentano azioni o comportamenti che si verificano all'interno della pagina web. È possibile monitorarli e modificarli tramite l'utilizzo dei **listener**, tramite il metodo **addEventListener()**, attivando funzion specifiche di callback quando si verifica un evento.

Anche gli **attributi HTMl** come, ad esempio **onclick** possono essere utilizzati per gestire alcuni eventi specifici come il click del mouse, ma si tratta di metodi obsoleti.

Eventi comuni:

- **click**
- **dblclick**: doppio click;
- **mousedown:** quando un pulsante del mouse viene premuto su un elemento;
- **mouseup:** quando un pulsante del mouse viene rilasciato su un elemento;

- **mouseenter:** quando il cursore del mouse entra in un elemento;
- **mouseleave**: quando il cursore del mouse esce da un elemento;
- **keydown:** quando viene premuto un tasto sulla tastiera;
- **keyup:** quando viene rilasciato un tasto sulla tastiera;
- **submit:** quando viene inviato un form;
- **load:** quando una risorsa viene caricata;
- **resize:** quando una dimensione della finestra viene modificata;

RILEGGERE BENE DISPENSA LEZIONE 45 CHE COMMENTA DEL CODICE POINT TO POINT

BOOTSTRAP

Framework front-end opensource sviluppato da Twitter; 2011, altamente personalizzabile.

https://getbootstrap.com

Per utilizzarlo in un progetto, basta scaricare il pacchetto e mettere la cartella "dist" nella cartella di progetto; dopo, è necessario settare i link del file .css e .js nel nostro codice ("dist/css/bootstrp.min.css" e "dist/js/bootstrap.min.js"). Altrimenti, posso utilizzare la versione CDN (content delivery network), per linkare direttamente online le due risorse senza bisogno di scaricare il pacchetto.

Ci stiamo concentrando quindi sul responsive, privilegiando i device mobile.

Viewport

L'area visibile di una pagina web all'interno del browser; si utilizza con il **metatag "viewport".**

Attributi di <viewport>:

- **width;**
- **height;**
- **initial-scale:** specifica il fattore di scala inziale quando la pagina viene caricata;
- **minimum -scale:** fattore di scala minimo;
- **maximum-scale:** fattore di scala massimo;
- **user-scalable:** specifica se l'utente può zoomare sulla pagina web;

I settaggi caratteristici sono:

- **width=device-width** -> imposta la larghezza a quella del dispositivo;
- **initial-scale=1.0** -> la pagina viene mostrata alla sua scala naturale;
- **minimum-scale=1.0** -> l'utente non può ridurre;
- **maximum-scale=1.0** -> l'utente non può ingrandire;
- **user-scalable=no**

Bootstrap è mobile-first, quindi il codice è ottimizzato per i device mobile.

Flexbox

È stato introdotto da CSS3 e offre una serie di proprietà che aiutano a definire il comportamento dei contenitori e degli elementi al loro interno.

Utilizza i due assi (orizzontale e verticale) per posizionare e distribuire gli elementi all'interno dei contenitori.

Proprietà di FlexBox:

- **display:** decide se il container deve essere gestito tramite flexbox o meno;
- **flex-direction:** definisce l'asse principale del container per la disposizione degli elementi; può essere **row, column, row-reverse e column-reverse;**
- **justify- content:** definisce la distribuzione sull'asse principale; può essere **flex-start** (allineamento all'inizio)**, flex-end** (allineamento alla fine)**, center** (allineamento al centro)**, space-between** (spaziatura tra gli elementi) **e space-around** (spaziatura attorno agli elementi)**;**
- **align-elements:** definisce come gli elementi debbano essere allineati rispetto all'asse trasversale; può essere **flex-start (**allineamento all'inizio), **flex-end** (allineamento alla fine), **center** (allineamento al centro), **stretch** (allineamento a tutta la lunghezza) e **baseline** (allineamento alla base del testo)**;**
- **flex-wrap:** questa proprietà definisce se gli elementi all'interno di un conteiner debbano essere disposti su una sola riga o su più righe; può essere **nowrap** (su una sola riga), **wrap** (su più righe) e **wrap-reverse** (con righe in ordine inverso);

display: flex prevede la forma contratta **d-flex.** *Es.*
 I'm a flexbox container!
In bootstrap, la classe <p> ad esempio sta per padding.

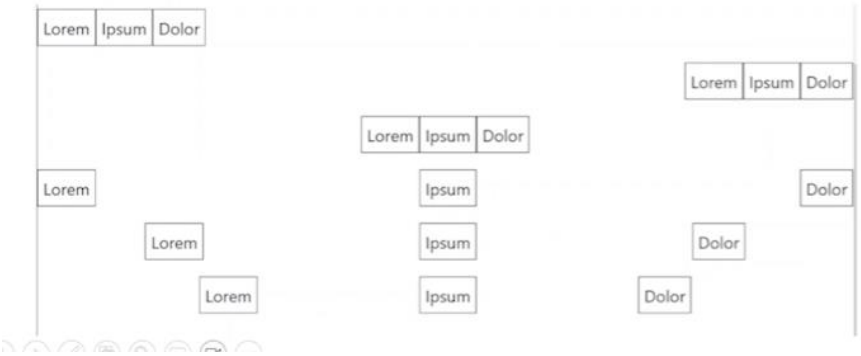
Esempi di visualizzazione:

```
<div class="d-flex flex-row mb-3">
  <div class="p-2">Flex item 1</div>
  <div class="p-2">Flex item 2</div>
  <div class="p-2">Flex item 3</div>
</div>
<div class="d-flex flex-row-reverse">
  <div class="p-2">Flex item 1</div>
  <div class="p-2">Flex item 2</div>
  <div class="p-2">Flex item 3</div>
</div>
```

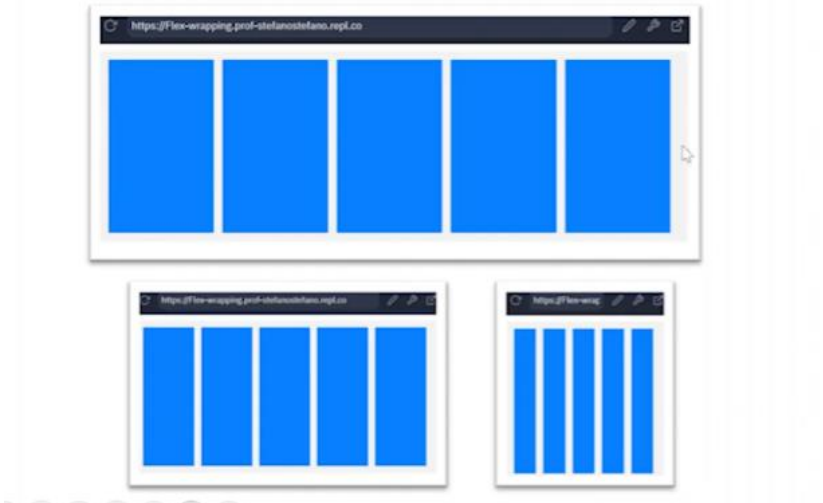
```
<div class="d-flex flex-column mb-3">
  <div class="p-2">Flex item 1</div>
  <div class="p-2">Flex item 2</div>
  <div class="p-2">Flex item 3</div>
</div>
<div class="d-flex flex-column-reverse">
  <div class="p-2">Flex item 1</div>
  <div class="p-2">Flex item 2</div>
  <div class="p-2">Flex item 3</div>
</div>
```

Flex item 1
Flex item 2
Flex item 3
Flex item 3
Flex item 2
Flex item 1

```
justify-content-start
justify-content-end
justify-content-center
justify-content-between
justify-content-around
justify-content-evenly
```

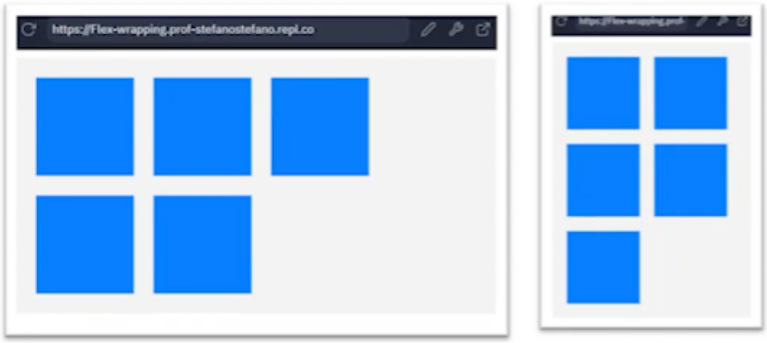


Nessun wrapping (predefinito dal browser) con `.flex-nowrap`



nessun wrapping e `flex-shrink: 0`

Con wrapping mediante `.flex-wrap`



Di default, il box model calcola le dimensioni di un elemento includendo solo altezza e larghezza e senza tenere in conto padding e bordo. Tuttavia, il browser calcolerà le dimensioni complessive dell'elemento quando usa box-sizing: border-box.

Ulteriori attributi utili:

- **flex-grow:** specifica quanto un elemento può espandersi rispetto ad altri elementi all'interno del container flessibile. Di default è 0, cioè l'elemento non si ingrandisce.
- **flex-shrink:** specifica quanto un elemento può ridursi rispetto ad altri elementi all'interno del container flessibile. Di default è 1, cioè l'elemento si ridurrà in maniera proporzionale agli altri elementi quando lo spazio si ridurrà.
- **flex-basis:** si specifica la dimensione base di un elemento flessibile prima che il contenitore flessibile distribuisca lo spazio disponibile.

Se si incontra un solo valore numerico flex, si sta impostando solo "flex-grow"; gli altri due valori rimangono di default.

Posso usare l'attributo **order: n** per cambiare l'ordine degli elementi flex:

```
.box-1 {flex: 1;order: 3;}
.box-2 {flex: 2;order: 2;}
.box-3 {flex: 1;order: 1;}
```

Lorem ipsum

Lorem Picsum (https://picsum.photos) -> genera foto casuali (prese dal catalogo del fotografo Ian Barnard)

Grid Layout

Sistema predefinito a **12 colonne** e righe fornito da bootstrap (da "col-1" a "col-12" come classi css).

i.e. se voglio creare uno spazio a 2 colonne, utilizzo "col-6" per tutte due come classe.

5 classi di breakpoint (consentono di definire il comportamento della griglia alle diverse larghezze di schermo).

- **.col-** dispositivi extra small – Larghezza schermo < 576px;
- **.col-sm-** dispositivi small – larghezza schermo >= 576px;
- **.col-md-** dispositivi medium – larghezza >= 768px;
- **.col-lg-** dispositivi large – larghezza >= 992px;
- **.col-xl-** dispositivi xlarge – larghezza >= 1200px;
- **.col-xxl-** dispositivi xxlarge – larghezza schermo >= 1400px;

	xs <576px	sm ≥576px	md ≥768px	lg ≥992px	xl ≥1200px	xxl ≥1400px
Container `max-width`	None (auto)	540px	720px	960px	1140px	1320px
Class prefix	`.col-`	`.col-sm-`	`.col-md-`	`.col-lg-`	`.col-xl-`	`.col-xxl-`
# of columns	12					
Gutter width	1.5rem (.75rem on left and right)					
Custom gutters	Yes					
Nestable	Yes					
Column ordering	Yes					

Posso concatenare più classi per adattare il layout a vari dispositivi:

```
<div class="container">
    <div class="row">
      <div class="col-sm-6 col-md-4 col-lg-3"
style="background-color: red;color: white;"">Colonna 1</div>
```

Nel caso siano presenti più colonne di quelle ospitabili con il layout scelto (i.e. div con col-sm-6 e 4 colonne su schermo minore di 576px), quelle in eccesso andranno sulla riga successiva.

W3Cschool Bootstrap 5 Tutorial

Navigation Bar

Esiste di default in bootstrap ed è personalizzabile; le classi sono **"navbar" e "navbar-expand-md";**

"navbar-dark" è la classe che indica che la barra è scura, e quindi il testo è chiaro.

```
<nav class="navbar navbar-expand-md navbar-dark bg-dark">
    <div class="container-fluid">
      <a class="navbar-brand" href="#">Il mio sito</a>
      <button class="navbar-toggler" type="button"
        data-bs-toggle="collapse" data-bs-target="#navbarNav"
        aria-controls="navbarNav" aria-expanded="false"
        aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse" id="navbarNav">
        <ul class="navbar-nav">
          <li class="nav-item">
            <a class="nav-link active" aria-current="page" href="#">Home</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="#">Chi siamo</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="#">Contatti</a>
          </li>
        </ul>
      </div>
    </div>
  </nav>
```

Il div **"container-fluid"** contiene il logo ed un pulsante di espansione; il pulsante è stato utilizzato con la classe **"navbar-toggler"** e l'attributo **"data-bs-target"** per specificare l'ID del div di destinazione che contiene la lista di navigazione.

La lista di navigazione è creata tramite la classe **"navbar-nav"** contiene 3 elementi, ognuno da un elemento della lista **"li"** e un link **"a".**

La barra viene resa **responsive** grazie alla classe "**navbar-expand-md**" che spiega al layout che deve espandersi sui device con schermo > 768px.

Componenti modali

Finestre pop-up che forniscono informazioni o permettono all'utente di interagire. Si può usare la classe **"modal"** e gli attributi **"data-bs-toogle"** e **"data-bs-target"** per attivarli; il contenuto dell'elemento è definito da un div con classe "**modal-content**" all'interno di un div con classe "**modal-dialog**".

Media queries

Consentono di definire regole CSS per dispositivi specifici; permettono di specificare stili diversi per situazioni diverse.

Interagiscono con il metatag **viewport** per creare siti responsive. Bootstrap utilizza le media queries per definire lo stile delle colonne di layout e le regole per definire i breakpoint.

Esempio:

Media queries

```
<style>
@media only screen and (min-width: 768px) {
.container {max-width: 800px;}
}
@media only screen and (min-width: 992px) {
.container {max-width: 1000px;}
}
@media only screen and (min-width: 1200px) {
.container {max-width: 1200px;}
}
</style>

<div class="container">

<div class="row">

<div class="col-sm-6 col-md-4 col-lg-3" style="background-
color: red;color: white;">Colonna 1</div>

<div class="col-sm-6 col-md-4 col-lg-3" style="background-
color: green;color: white;">Colonna 2</div>

<div class="col-sm-6 col-md-4 col-lg-3" style="background-
color: blue;color: white;">Colonna 3</div>

<div class="col-sm-6 col-md-4 col-lg-3" style="background-
color: black;color: white;">Colonna 4</div>

</div>

</div>
```

Components

Button: Alcuni predefiniti:

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>
<button type="button" class="btn btn-link">Link</button>
```

Caroselli (o slider): sono un carosello (classe **"carousel"**) di immagini e/o elementi multimediali (class "**carousel-item**"), con tanto di bottoni per scorrere ("**carousel-control-prev/next**", mentre la proprietà "**data-bs-slide**" specifica la direzione di rotazione) o scorrimento automatico. Viene reso responsive con la classe "**w-100**" che imposta la larghezza dell'immagine al 100% della larghezza del container padre.

Toast: viene utilizzato per inviare messaggi temporanei (alert) all'utente. Sono posizionati in un angolo della pagina e spariscono dopo un tot di tempo.

Progress bar: div class="progress", posso settare il valore e poi creare script che la utilizzino.

Spinner: animazione di una icona che serve a rappresentare una certa azione come il caricamento.

List-group: mostra un elenco di elementi in modo stilizzato; formate da una serie di div e link racchiuse in un contenitore "list-group"; ogni elemento è un "list-group-item". Personalizzabile tramite, ad esempio, "list-group-flush".

Pagination: crea una serie di pagine numerate con pulsanti di scorrimento. È formata da una **"ul"** con la classe **"pagination"** e di una serie di **"li"** e dei **"tabindex"** che definiscono l'indice.

JQuery

2006 John Resig, lib che semplifica la scrittura del codice attraverso una manipolazione semplificata del DOM e una interazione semplificata con HTML.

Lo posso includere utilizzando una CDN (Content Delivery Network) , con il link diretto alla libreria sul sito di jQuery oppure scaricandola scegliendo tra varie versioni:

- **compressed** (ottimizzata per la produzione);
- **uncompressed** (contiene tutti i commenti e la formattazione del codice, meno agile in produzione);
- **minify** (compressa e con gli spazi vuoti rimossi);
- **slim version** (versione ridotta che comprende solo le funzioni più utilizzate);

Esempio di codice:

```
<!DOCTYPE html>
<html>
<head>
<title>Esempio jQuery</title>
<script src="https://code.jquery.com/jquery-3.6.0.min.js">
</script>
<script>
$(document).ready(function() {
        $("h1").text("Hello world!");
});
</script>
</head>
<body>
        <h1>Questo è un titolo</h1>
        <p>Questo è un paragrafo</p>
</body>
</html>
```

Lo script è nell'head, quindi viene eseguito subito, ma lo stato $(document).ready(function {..}) esegue la funzione solo quando il documento è caricato. In questo caso, al posto di "Questo è un titolo", apparirà "Hello World". -> selettore = $("h1") metodo = .text

Sintassi

$selettore.metodo(valore);

- **$:** indica l'utilizzo di JQuery;
- **selettore:** è un'istruzione che indica l'elemnto HTML su cui eseguire un'azione;
- **metodo:** è l'azione che vogliamo eseguire sull'elemento selezionato;
- **valore:** è un parametro opzionale che possiamo utilizzare per fornire ulteriori info sull'azione che eseguiamo;

Esempi:

$("p").hide(); nasconde tutti i paragrafi;

JQuery supporta le funzioni di **callback**, che permette di eseguire funzioni dopo che una azione è stata completata (i.e**. .fadeIn(numero millisecondi, funzione da eseguire**)) .

Tipi di selettori:

- **Selettore di Tipo:** seleziona tutti gli elementi di un tipo (i.e. <p> o <h1>)
- **Selettore di ID:** seleziona un elemento in base all'id (i.e. "#mio_id");
- **Selettore di classe:** seleziona tutti gli elementi con una classe (i.e. ".mialista");
- **Selettore di attributo:** $("[name="mio_nome"]");
- **Selettore di figlio:** $("ul li") seleziona tutti gli elementi <li> figli di un elemento <ul>;
- **Selettore di primo figlio:** $("ul li: fisrt-child");
- **Selettore di ultimo figlio:** $("ul li: last-child");
- **Selettore di primo ed ultimo elemento:** $("p:first") primo paragrafo e $("p:last") ultimo paragrafo;

Eventi

Esempi:

click: $(elemento).on(“click”, funzione)

hover: viene eseguito quando si posiziona il mouse sopra o fuori da un elemento ($(elemento).on(“mouseenter” / “mouseleave”, funzione))

e poi keypress, submit, change (quando cambia il valore di un elemento), resize…

VANTAGGI: Sintassi semplificata, alta compatibilità con i browser, cross-browser, ampia gamma di plugin e community molto attiva.

JQuery incude la sintassi per le funzioni condizionali ed i cicli.

Ajax

Chiamate sincrone: una chiamata che blocca il thread finché non riceve risposta;

Chiama asincrona: la chiamata non blocca il thread in attesa della risposta;

AJAX (Asynchronous JavaScript and XML) è una tecnica di programmazione che consente di **eseguire richieste http asincrone tramite JS;** permette di creare interfacce utente dinamiche e reattive senza attendere il caricamento di tutta la pagina web;

Esistono metodi nativi, quali **XMLHttpRequest e Fetch API** oltre al altre librerie quali jQuery, Axios, SuperAgent…

- Flusso di lavoro di una richiesta Ajax;
- Evento nella pagina web;
- Chiamata server remoto con creazione di un oggetto **XMLHttpRequest;**
- elaborazione della richiesta da parte del server e risposta da parte sua;
- Letture della risposta da parte di JS;
- Aggiornamento della pagina web;

XMLHttpRequest e Fetch API

Implementati direttamente nel browser.

XHR: nata prima, più complessa;

FetchAPI: posteriore e più semplice; introdotta nel 2015 (JavaScript ES6), utilizza le **promises.**

Le promises sono delle operazioni asincrone che potrebbero non essere ancora completate e consentono di gestire il risultato appena disponibile, limitando l’utilizzo di callback annidate.

```
// XHR
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://example.com/data.json');
xhr.onload = () => {
  if (xhr.status === 200) {
    const data = JSON.parse(xhr.responseText);
    console.log(data);}
};
xhr.send();
```

```
// FETCH API
fetch('https://example.com/data.json')
  .then(response => {
    if (response.ok) {
      return response.json();
    }
    throw new Error('Network response was not ok.');
}).then(data => console.log(data))
.catch(error => console.error(error));
```

DIFFERENZE:

XHR	FetchAPI
È necessario creare un'stanza dell'oggetto XMLHttpRequest, aprire una connessione verso l'URL specificato, impostare una funzione di Callback per gestire la risposta e inviare la richiesta;	La funzione fetch restituisce una Promise che rappresenta la riposta http alla richiesta effettuata; quando la promise viene risolta, il metodo then viene chiamato e riceve in input un oggetto Response, che rappresenta la risposta http.
La risposta viene gestita nella funzione di callback impostata tramite onload;	Il primo blocco controlla se la riposta http ha avuto successo (response.ok restituisce true se la risposta ha uno stato http 2xx). In caso di successo viene chiamato il metodo json() sulla risposta Response. Per ottenere il corpo della risposta in formato JSON, che viene restituito con una nuova Promise.
La gestione degli errori viene effettuata nella funzione di callback impostata tramite onerror;	Il secondo blocco riceve in input il risultato della Promise restituita dal metodo json(), ovvero il corpo della riposta in formato JSON.
	Se si verifica un errore durante la richiesta http o durante il parsing del corpo della risposta in formato json, viene lanciata una eccezione e la promise viene rifiutata. Il blocco Catch viene chiamato e riceve in input l'errore che viene stampato dalla console

jQuery e Ajax

jQuey offre un metodo dedicato $.ajax() che semplifica la creazione di richieste AJAX personalizzate.

Esempio di richiesta GET e promessa di ricevere risposta JSON:

```
$.ajax({
  url: 'https://example.com/data.json',
  method: 'GET',
  dataType: 'json',
  success: function(data) {
    console.log(data);
  },
  error: function(jqXHR, textStatus, errorThrown) {
    console.error(textStatus, errorThrown);
  }
});
```

Ci sono altri metodi per le chiamate asincrone, quali **$.get:**

```
$.get('https://example.com/data.json',function(data)
{
  console.log(data);
});
```

$.post:

```
$.post('https://example.com/data',
  {name:'John',age:30}, function(data) {
  console.log(data);
});
```

$.getJSON (richiede solo l'Url della risorsa da richiedere e una funzione di callback per gestire il corpo della risposta JSON)**:**

```
$.getJSON('https://example.com/data.json',
  function(data) {
  console.log(data);
});
```

$.ajaxSetup (permette di impostare opzioni di default per **tutte** le richieste AJAX effettuate con jQuery)**:**

```
$.ajaxSetup({
  dataType: 'json',
  beforeSend: function(xhr) {
    xhr.setRequestHeader(
        'Authorization',
        'Bearer ' + token);
  }
});
```

CORS

Sistema utilizzato dai browser per prevenire gli attacchi XSS; prevede che un sito web possa accedere alle risorse di un altro solo se quest'ultimo esplicitamente consente l'accesso attraverso l'utilizzo di **apposite header http.** Possiamo configurare il server destinazione per inviare appositi header http nelle richieste AJAX provenienti da altri domini come ad esempio l'header '**Access-Control-Allow-Origin**' con l'elenco dei domini da cui accettare connessioni.

- Esistono altri header, quali **'Access-Control-Allow-Methods'** e **'Access-Control-Allow-Headers'** per specificare i metodi http e i tipi di dati accettati dalle richiese AJAX provenienti da altri domini.
- Potremmo invece utilizzare un **proxy server** per effettuare le richeste AJAX ed evitare le restrizioni del CORS.
- Oppure usare JSONP (JSON With Padding); JSONP prevede l'utilizzo di una callback per ricevere i dati JSON dal server, effettuando una richiesta AJAX tramite un tag script HTML.
  Quando viene effettuate una chiamata AJAX JSONP il browser crea un tag script HTML con l'url della risorsa da richiedere; in esso viene specificato una callback, ossia una func JS che verrà chiamata quando il server di destinazione restituirà i dati JSON.
  Il server restituisce i dati json incapsulati nella funzione specificata nella callback.
  Quando il browser riceve la risposta, esegue la func JS specificata nella callback passando i dati JS come parametro.

TEST CAPITOLO 55 DA RIVEDERE

Node.js

Piattaforma di sviluppo per applicazioni web in JS.

Client-server, modalità asincrona e non bloccante. Permette di creare server web che permettono di gestire sia richieste http che webSocket; utlizza gli **NPM (Node Packet Manager)**, pacchetti che permettono di gestire ed espandere le varie potenzialità (i.e. Express o Socket.io).

Caratteristiche:

- **Asincrono:** l'architettura asincrona permette di creare un singolo thread per gestire tutte le richieste in ingresso senza bloccare il thread principale, fornendo una user-experience più reattiva;
- **Prestazioni elevate** grazie all'uso di JavaScript V8;
- **Modulare:** estendibile anche tramite funzionalità e librerie di terze parti;
- **Facile da sviluppare**
- **Scalabile**

Pacchetto installabile contenente motore runtime e gestore dei moduli NPM.

Da CMD:

node -v

Per creare una nuova app posso usare il comando ***npm init***; questo crea un file package.json che contiene la struttura e le info di base (metadati, dep etc..).

Struttura di un pacchetto json:

```
{
  "name": "my-app",
  "version": "1.0.0",
  "description": "Descrizione del mio progetto",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

La voce scripts indica quali "scripts" sono inclusi nel progetto, esattamente come quella "dependencies".

A questo punto, devo installare i pacchetti necessari con **npm install <nomepacchetto>**; posso includere i pacchetti nel codice così:

```
const nomepacchetto = require('nomepacchetto');
const app = nomepacchetto();
```

Ricordarsi sempre di verificare la variabile di sistema PATH sia settata per nodejs.

Per eseguire un pacchetto creato, bisogna utilizzare la sintassi **npm run <nomepacchetto>**

In alcuni casi, anche da js è possibile effettuare una operazione in modo sincrono come ad esempio:

```
const fs = require('fs');
const data = fs.readFileSync('/path/to/file', 'utf-8');
console.log(data);
```

Tuttavia, potrei gestire la stessa operazione in modalità asincrona:

```
const fs = require('fs');
fs.readFile('path/to/file', 'utf-8', (err, data) => {
if (err) throw err;
console.log(data);

    });
```

In node.js, le operazioni di I/O sono sempre asincrone, per non bloccare il thread di esecuzione del programma.

In JS esiste un costrutto (**async/await**), introdotto in ES2017 (ES8) che facilità la progrmmazione asincrona:

- **async: utilizzato per dichiarare funzioni asincrone, cioè func che restituiscono una promise**. Al loro interno, posso utilizza **await** per attendere il completamento di un'altra funzione asincrona o di una promise.
- **Await: sospende l'esecuzione della funzione asincrona in attesa del completamento della promise**; se la promise viene risolta, ne restituisce il valore, se viene rifiutata solleva una eccezione**.**

Esempio:

```
async function getHello() {
  const response=await fetch('https://api.example.com/hello');
  const data = await response.json();
  return data.message;
}

getHello().then(message => {
  console.log(message);
});
```

La funzione attende (con await) il fetch dell'url (promise) per fare il parsing del risultato con response.json().

WebSockets

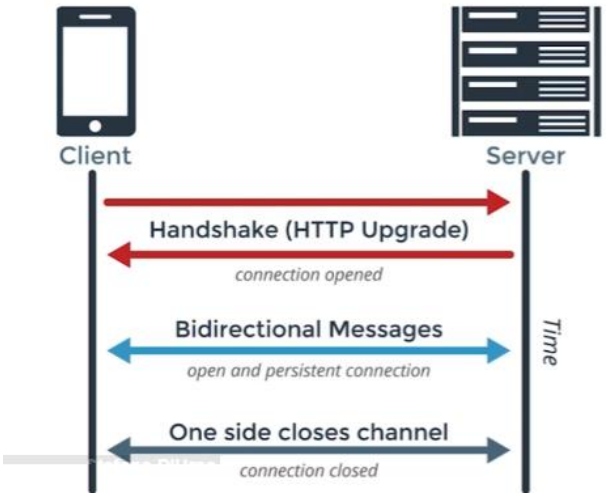
Tecnologia di comunicazione bidirezionale/server per lo scambio di informazioni in metodo efficiente e scalabile**. A differenza di http, che utilizza una singola richiesta-risposta, i websockets sono bidirezionali e costanti.**

Hanno un protocollo specifico che viene stabilito tramite **http handshake.**

Lato client: si utilizza la classe "WebSocket" inclusa nel browser.

Lato server: si utilizza una libreria specifica, come "ws" per NodeJs.

Una volta stabilita la comunicazione, si possono utilizzare i metodi come "send" della classe WebSocket lato client che il metodo "send" dell'oggetto di connessione lato server.



*La **richiesta client,** di tipo http-GET, presenta un campo "Upgrade" nell'header impostato su "WebWocket"; il campo "Connection" dell'header viene impostato su "Upgrade" e il campo "Sec-WebSocket-Key" contiene un valore random generato dal client.*

*Il **server** risponde con una risposta di handshake WebSocket via http. Il campo "Upgrade" viene impostato su "websocket", il campo "Connection" su "Upgrade" e il campo "Sec-WebSocket-Accept" contiene un valore derivato dal campo "Sec-WebSocket-Key" della richiesta client di handshake.*

Il valore del campo "Sec-WebSocket-Accept" è generato tramite func di hash SHA-1, una costante predefinita ed il valore del campo "Sec-WebSocket-Key" della richiesta di handshake del client.

Il protocollo WebSocket prevede di inviare messaggi, dati di tipo binario o di testo, in singolo frame o in più frame (utilizzando un frame di intestazione per descriverne la lunghezza); il frame di intestazione contiene info sul messaggio, il payload rappresenta il corpo del messaggio stesso.

WebSocket prevede la gestione degli errori.

Socket.IO

Libreria JS utilizzata per gestire la comunicazione web in tempo reale; fornisce un layer di astrazione superiore ai WebScokets. Utilizza una serie di tecniche per fornire una connessione stabile ed affidabile tra client e server, semplificando la comunicazione. Supporta numerosi framework, browser e device.

Ovviamente, è necessario settare un server Socket.IO che ascolti la connessione ed includere la libreria lato client.

Implementazione

```
const express = require('express');
const WebSocket = require('ws');
const app = express();

const server = app.listen(3000, () => {
  console.log('Server web avviato');
});

const wsServer = new WebSocket.Server({ server });
wsServer.on('connection', (socket) => {
  console.log('Client connesso');
  socket.on('message', (message) => {
    console.log('Messaggio ricevuto:', message);
    // Invia il messaggio di risposta al client
    socket.send(`Hai detto: ${message}`);
  });
  socket.on('close', () => {
    console.log('Client disconnesso');
  });
});
```

Lato Client:

```
// Crea il client WebSocket
const socket = new WebSocket('ws://localhost:3000');

socket.onopen = () => {
  console.log('Connesso al server WebSocket');
  socket.send('Ciao server WebSocket!');
};

socket.onmessage = (event) => {
  console.log('Messaggio ricevuto dal server:', event.data);
};

socket.onclose = () => {
  console.log('Connessione al server WebSocket chiusa');
};
```

Posso usare il server node.js su replit inserendo a console:

node index.js

Tuttavia, replit usa HTTPS come protocollo definito; quindi, per utlizzare WebSockets, ho due scelte:

1) Creare un cert SSL dedicato
2) Utilizzare una risorsa esterna, come socket.io

Esempio della implementazione di socket.io:

index.js

```
io.on('connection', (socket) => {
  console.log('Nuova connessione');

  // Ricezione del messaggio dal client
  socket.on('message', (message) => {
    console.log('Messaggio ricevuto:', message);

    // Invio del messaggio di risposta all client
    socket.emit('message', `Hai detto: ${message}`);
  });

  socket.on('disconnect', () => {
    console.log('Client disconnesso');
  });
});
```

Index.html

```
<script src="/socket.io/socket.io.js"></script>

<script>
  const socket=io(
        'wss://socketiotest.prof-stefanostefano.repl.co');

  // Gestione dell'invio del messaggio dal form
  const form = document.getElementById('form');
  form.addEventListener('submit', (event) => {
    event.preventDefault();
    const messageInput = document.getElementById('message');
    const message = messageInput.value;
    socket.emit('message', message);
    messageInput.value = '';
  });
</script>
```

```
<script src="/socket.io/socket.io.js"></script>

<script>
  // Gestione del messaggio di risposta dal server
  socket.on('message',(message)=>{
    const messagesContainer=document.getElementById('messages');
    const messageElement = document.createElement('p');
    messageElement.textContent = message;
    messagesContainer.appendChild(messageElement);
  });
</script>
```

DA RIVEDERE BENE PER FARE IL TEST (CAP 58, ultimo Video)

Node js – Express

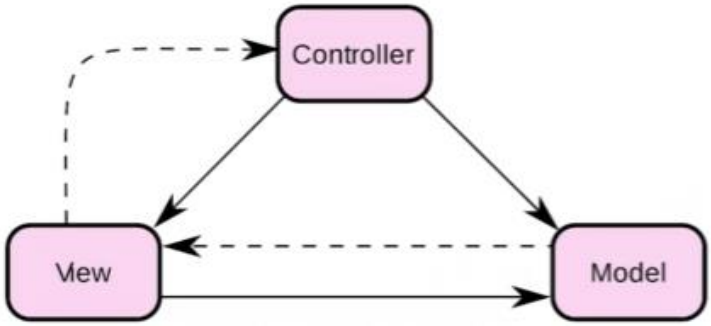
Framework per la creazione di webapp;

Funzionalità:

- **gestione delle richieste** http (GET, POST, PUT, DELETE);
- **definizione delle rotte:** gestisce l'ingresso di risorse e richieste tramite la sintassi ***app.METHOD(PATH, HANDLER)***, dove METHOD è il metodo http, PATH è il percorso della rotta e HANDLER è una funzione che gestisce la richiesta;
- **implementazione di middleware** ovvero funzioni che vengono eseguite prima o dopo la gestione delle richieste http, quali autenticazione, log etc...
- **utilizzo di template engine** come handlebars, EJS e Pug;

MVC

Pattern architetturale nato per separare le responsabilità del codice (Model-View-Controller):



Model: i dati dell'applicazione e le operazioni che si svolgono su di essi; non consce view e controller, fornisce solo una interfaccia per accedere ai dati.

View: interfaccia utente che visualizza i dati forniti da Model. La view non conosce model e controller, ma fornisce solo una interfaccia utente per interagire con l'app.

Controller: coordina l'interazione tra model e view; riceve le richieste da view, accede ai dati del model e aggiorna la view con i nuovi dati.

Si può implementare tramite Express.

Il **model** può essere implementato tramite librerie quali Mongoose, Sequelize o Knex.js.

La **view** può essere implementata tramite template engine quali Handlebars, EJS o Pug.

Il **controller** può essere implementato tramite funzionalità di routing di express.js.

Esempio di struttura di path in applicativo MVC:

```
node-express-app/
  |- controllers/
  |     |- userController.js
  |     |- postController.js
  |- models/
  |     |- userModel.js
  |     |- postModel.js
  |- views/
  |     |- home.hbs
  |     |- user.hbs
  |     |- post.hbs
  |- public/
  |     |- css/
  |     |- js/
  |- routes/
  |     |- index.js
  |     |- userRoutes.js
  |     |- postRoutes.js
  |- app.js
  |- package.json
  |- README.md
```

Esempio di implementazione:

```
// Importa le dipendenze necessarie
const express = require('express');
const bodyParser = require('body-parser');
const path = require('path');

// Crea un'istanza dell'applicazione Express
const app = express();

// Middleware per il parsing del corpo delle richieste HTTP
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

// Configura la cartella "public" per i file statici
app.use(express.static(path.join(__dirname, 'public')));


// Definisci le rotte dell'applicazione
const indexRouter = require('./routes/index');
const userRouter = require('./routes/user');
const postRouter = require('./routes/post');
app.use('/', indexRouter);
app.use('/users', userRouter);
app.use('/posts', postRouter);

// Gestisci gli errori
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Internal server error');
});

// Avvia il server sulla porta specificata
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server avviato sulla porta ${PORT}`);
});
```

Rotte

Definiscono come l'app gestisce le richieste http che provengono dal client; possono essere stringhe, regexp o combinazioni delle due.

Le rotte possono essere definite per i diversi metodi http, magari utilizzando file separati per mantenere pulito il codice. **Una volta definiti i router che gestiscono le rotte per un gruppo di funzionalità, le si può collegare all'app principale con il metodo app.use().**

Esempio:

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

app.get() definisce la rotta per le richieste http GET dell'url /users. La func res.send() invia una risposta http con il messaggio "Hello World!".

Potrei utilizzare res. anche per gestire le callback; potrei anche utilizzare req.params per accedere ai dati di una richiesta e quindi ai parametri di una URL sempre in callback.

Es.

```
router.get('/:id', (req, res) => {
  const myId = req.params.id;res.send(`Hello World ${myId}`);
```

Alla fine di ogni file che contiene delle routes è importante prevedere una funzione che esporti le stesse per renderle al file che gestisce l'app:

```
const express = require('express');
const router = express.Router();

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

router.get('/:id', (req, res) => {
  const myId = req.params.id;res.send(`Hello World ${myId}`);
});

router.post('/', (req, res) => {
  const _body = req.body;res.send('POST');
});

module.exports = router;
```

Questo codice definisce tre rotte per il router: GET all'url "/", GET all'url "/:id", POST all'url"/".

Le rotte definite utilizzano il metodo res.send() per inviare la risposta http al client come messaggio di testo.

Il parametro dinamico :id viene usato per "catturare" un valore dinamico dell'URL, tramite la proprietà req.params().

Nella rotta POST, la richiesta è gestita tramite req.body per recuperare il corpo della richiesta http; è necessario che il middleware "body-parser" sia configurato per gestire al meglio questa funzionalità.

MVC Views

Module

Un modulo è un file che contiene codice, variabili, funzioni o oggetti che possono essere utilizzati in altri moduli. La variabile module.exports è una proprietà dell'oggetto module che viene utilizzata per esportare funzioni, oggetti, variabili o altre cose presenti nel modulo stesso.

Es.

File myModule

```
// Definiamo la funzione
function myFunction() {
  console.log('Hello, world!');
}

// Esportiamo la funzione tramite module.exports
module.exports = myFunction;
```

Altro file

```
// Importiamo la funzione dal nostro modulo
const myFunction = require('./myModule');

// Utilizziamo la funzione
myFunction(); // Output: "Hello, world!«
```

è possibile **esportare anche singole funzioni:**

module.exports = funzione1, funzione2;

Oppure posso **esportare direttamente un oggetto**:

```
const myModule = {
    method1() { console.log('method 1'); },
    method2() { console.log('method 2'); },
    method3() {
        myModule.method2();
        console.log('method 3'); }
};
module.exports = myModule;
```

Oppure **esportare una classe:**

```
class MyClass {
    constructor() {
        this.myProperty = 'Hello world';
    }
    myMethod() {
        console.log(this.myProperty);
    }
}
module.exports = MyClass;
```

Views

In Express posso usare vari motori di engine, come EJS (Embedded Javascript). Esso implementa codice JS in pagine HTML; permette di definire modelli di pagina che vengono compilati dinamicamente con i dati dell'app.

EJS viene installato tramite NPM con *npm install ejs*. Successivamente, bisogna configurare l'app per usare EJS come motore per il template:

```
const express = require('express');
const app = express();
app.set('view engine', 'ejs');
```

Sarà poi necessario creare una pagina con estensione .ejs che definisce la struttura della pagina. Esso conterrà i tag HTML, CSS e JS oltre al tag EJS per incorporare i dati dell'applicazione.

*Esempio di tag tipico EJS (**<% name %>**):*

```
<!DOCTYPE html>
<html>
  <head>
    <title>Saluto</title>
  </head>
  <body>
    <h1>Ciao, <%= name %>!</h1>
  </body>
</html>
```

Uno dei modi più veloci è quello di **creare una rotta che renderizza la view.**

Es.

app.get('/hello/:name', (req, res) => {

const name = req.params.name;

res.render('hello', { name });

});

Il metodo **res.render('nomefile', {oggetto1, oggetto2…})** viene utilizzato per renderizzare la view e inviare la pagina HTML compilata al client.

La sintassi EJS <% … %> esegue il codice JS mentre la sintassi <%= … %> visualizza il valore di una variabile all'interno della pagina.

Ulteriore esempio:

```
const myformRouter = require('./routes/myform');
app.use('/myform', myformRouter);
```

A questo punto occorre inserire un nuovo file myform.js all'interno di routes/

```
const express = require('express');
const bodyParser = require('body-parser');
const router = express.Router();
router.use(bodyParser.urlencoded({ extended: false }));

router.get('/', (req, res) => {
  res.render('form');
})
router.post('/', (req, res) => {
  const name = req.body.name;
  const email = req.body.email;
  const message = req.body.message;
  res.render('success', { name, email, message });
});
module.exports = router;
```

File form.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form</title>
  </head>
  <body>
    <h1>Form</h1>
    <form action="/myform" method="POST">
      <label for="name">Name:</label>
      <input type="text" name="name" required>
      <br>
      <label for="email">Email:</label>
      <input type="email" name="email" required>
      <br>
      <label for="message">Message:</label>
      <textarea name="message" required></textarea>
      <br>
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

form.ejs

File success.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title>Success</title>
  </head>
  <body>
    <h1>Success</h1>
    <p>Thank you for your submission, <%= name %>!</p>
    <p>We will contact you at <%= email %> soon.</p>
    <p>Your message was: <%= message %></p>
  </body>
</html>
```

Controller

Gestisce la logica del business; in Node.js posso implementare il controller attraverso Express; il controller si occupa sempre di ricevere le richieste provenienti dal client, accedere ai dati necessari per elaborare la richiesta e restituire una risposta al client stesso.

Esempio di codice:

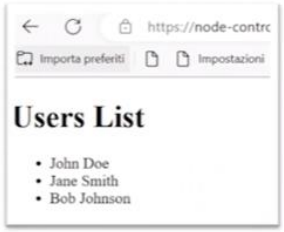
```
function listUsers(req, res) {
  const users = [
    { id: 1, name: 'John Doe' },
    { id: 2, name: 'Jane Smith' },
    { id: 3, name: 'Bob Johnson' },
  ];
  const filteredUsers = filterUsers(users, req.query.search);
  res.render('users', { users: filteredUsers });
}

function filterUsers(users, search) {
  if (!search) {
    return users;
  }
  return users.filter(user => {
    return
user.name.toLowerCase().includes(search.toLowerCase());
  });
}
module.exports = {listUsers};
```

usersController.js

```
const express = require('express');
const app = express();

const usersController=require('./controllers/usersController');

app.get('/users', usersController.listUsers);

app.set('views', './views');
app.set('view engine', 'ejs');

app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

app.js

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Users List</title>
</head>
<body>
  <h1>Users List</h1>
  <ul>
    <% users.forEach(user => { %>
      <li><%= user.name %></li>
    <% }); %>
  </ul>
</body>
</html>
```



users.js

La **route** è l'endpoint (URL) dell'app (nell'esempio sopra, la stringa app.get associa la route /users alla func listUsers() del controller usersController), il **controller invece è il componente che si occupa di gestire la logica di business dell'app** (nel nostro esmepio, si tratta di userController, che gestice le route attraverso la func listUsers()). **In Express quindi i due concetti sono fortemente correlati.**

Model

È la struttura dei dati che saranno gestiti da una app (i.e. i dati presenti in un database). **Spesso utilizza strutture NoSQL (ie. MongoDB) tramite le loro implementazioni (in Extpress, Mongoose).**

Esempio di codice:

```
[
  {"id": 1,"name": "john","nick": "johnny"},
  {"id": 2,"name": "jane","nick": "jany"},
  {"id": 3,"name": "bob","nick": "bobby"}
]
```

data/users.json

Implemento in index.js una **route** specifica che restituisca il nostro db:

```
app.get('/api/users', (req, res) => {
  const users = require('./data/users.json');
  res.json(users);
});
```

Poi, in userController gestico la richiesta di acquisizione degli utenti usando il metodo *usersFromDB:*

```
const database = require('../data/database');

// file usersController.js
const usersController = {
  registerUser(req, res) {[…]},

  filterUsers(users, search) {[…]},

  listUsers(req, res) {[…]},

  async usersFromDB(req, res) {
    const users = await database.getUsers();
    console.log(users);
    res.render('users', { users: users });
  },
};

module.exports = usersController;
```

userController.js

La funzione è richiamata tramite modulo di funzioni dedicato, come specificato in apertura di userController.js:

```
async function getUsers() {
  const response=await fetch('http://localhost:3000/api/users');
  const data = await response.json();
  return data;
}

async function saveUser(user) {
  // implementazione per salvare l'utente su un endpoint
  console.log("SAVEUSER");
}

module.exports = {
  getUsers,
  saveUser,
};
```

data/database.js

Bisogna inserire nell'index.js a questo punto questo endPoint:

app.get('/dbusers', usersController.usersFromDB);

Per la fetch in getUsers() viene usata esplicitamente async/await e non una promise con then per evitare problemi di sincronia.

Middleware

Funzione che può accedere all'oggetto request e all'oggetto response di una app e può anche modificare il flusso di esecuzione delle richieste http. Lo possiamo utlizzare per validazione dei dati, autenticazione, compressione elle risposte, logging etc...

I middleware in Node.JS sono eseguiti **in cascata.** In caso di middleware **applicativi**, essi vengono eseguiti per **tutte le richieste http**, mentre per middleware di **route**, essi vengono **eseguiti solo per le route interessate; essi ricevono 3 parametri:**

- oggetto di richiesta (**request**);
- oggetto risposta (**response**);
- la funzione **next**;

La funzione next() viene utilizzata per passare il controllo al prossimo middleware nella catena di gestione delle risorse.

Esempio di middleware che controlla se l'utente ha effettuato l'accesso o meno:

```
function isAuthenticated(req, res, next) {
  if (req.session && req.session.user) {
    return next();
  } else {
    return res.redirect('/login');
  }
}
```

Potrei, come già detto, incorporarlo anche nelle route:

```
app.get(
        '/dashboard',
        isAuthenticated, function(req, res) {
   // qui si arriva solo se l'utente è autenticato
   res.render('dashboard', { user: req.session.user });
});
```

In questo caso, il middleware isAuthenticated viene utilizzato come secondo parametro della func app.get() in modo che venga eseguito prima dell'handler che renderizza la pagina di dashboard function(req,res)

Passport

Middleware di autenticazione in Node.js; i suoi **moduli** definiscono come verrà gestita l'autenticazione dai diversi **provider.**

Solitamente, la richiesta di autenticazione viene gestita tramite chiamata alla funzione di Passport **authenticate**, che riceve in input le info di autenticazione dell'utente e restituisce un callback che viene chiamato alla fine del processo di autenticazione.

Esistono varie strategie di autenticazione; analizziamone alcune:

- **Local authentication strategy:** utilizza per autenticare gli utenti di un sistema ad autenticazione locale (i.e. ad esempio un database di utenti).
- **Token-based authentication strategy:** utilizzata per autenticare utenti tramite token, ad esempio Json Web Token (JWT).
- **OAuth authteenticacion strategy:** utilizza OAuth, un protocollo di autorizzazione che consente agli utenti di autorizzare l'accesso alle loro risorse da parte di terze parti. Ci deve essere un token OAuth valido per verificare l'identità di un utente.

- **OpenID authentication strategy:** utilizza OpenID, un protocollo di autenticazione che consente agli utenti di autenticarsi utilizzando le loro credenziali di accesso di un provider di autenticazione. Ci deve essere un token OpenID valido per verificare l'identità di un utente.

Passport **serializza** (cioè li mette "in file" e li converte in un determinato foramto) i dati presenti in un oggetto per memorizzarli su una memoria persistente (DB, cookie etc..). Per recuperarli, gli stessi andranno **deserializzati.** Per fare questo, passport usa due funzioni: **serializeUser e deserializeUser.**

Per installare passport *npm install passport passport-local*.

Nella mia app devo importare il modulo e inizializzarlo; posso farlo nel file principale (app.js o server.js):

const passport = require('passport');

const app = express();

app.use(passport.initialize());

Dovrò poi configurare le strategie di autenticazione che ci interessano:

```
const LocalStrategy = require('passport-local').Strategy;
const User = require('./models/user');
 passport.use(new LocalStrategy(
  function(username, password, done) {
    User.findOne({ username: username }, function (err, user) {
      if (err) { return done(err); }
      if (!user) { return done(null, false); }
      if (!user.verifyPassword(password)) {
        return done(null, false); }
      return done(null, user);
    });}
));
```

La funzione sopra, presi in input user e pass, cerca un user nel db e restituisce le varie risposte di login; **sarà poi necessario creare le funzioni di serializzazione e deserializzazione:**

```
passport.serializeUser(function(user, done) {
  done(null, user.id);
});

passport.deserializeUser(function(id, done) {
  User.findById(id, function (err, user) {
    done(err, user);
  });
});
```

Potrei poi utilizzare Passport per gestire l'autenticazione delle rotte dell'app:

```
app.post(
  '/login',
  passport.authenticate('local',
  { successRedirect: '/',failureRedirect: '/login' }));
```

Passport e OAuth

Analizziamo il caso specifico di utilizzare OAuth con Facebook per una web-app:

npm install passport-facebook dotenv

Bisogna poi creare una nuova app su facebook developer e configurare le credenziali dell'applicazione; questo serve per configurare correttamente le callback url dell'app, che Passport utilizzerà per gestire l'autenticazione con Facebook.

Configuro poi Passport e definisco la strategia di autenticazione OAuth con Facebook:

```
const passport = require('passport');
const FacebookStrategy=require('passport-facebook').Strategy;

passport.use(new FacebookStrategy({
    clientID: process.env.FACEBOOK_CLIENT_ID,
    clientSecret: process.env.FACEBOOK_CLIENT_SECRET,
    callbackURL: "/auth/facebook/callback"},
  function(accessToken, refreshToken, profile, done) {
    // Verificare l'identità dell'utente e restituirlo
    return done(null, profile);
  }
));
```

La func done() verifica l'identità dell'utente e restituisce l'oggetto profile.

Si definiscono poi le rotte di autenticazione:

```
app.get('/auth/facebook', passport.authenticate('facebook'));
app.get('/auth/facebook/callback',
  passport.authenticate('facebook',{failureRedirect:'/login'}),
  function(req, res) {
    // L'autenticazione con Facebook ha successo, reindirizza
l'utente alla pagina principale dell'applicazione
    res.redirect('/');
  });
```

La prima rotta riguarda l'url per l'avvio dell'auth; la seconda il callback per l'url specificata prima e utilizza passport per gestire l'autenticazione con FB. Se ha successo, l'utente viene redirezionato alla pagina principale dell'app.

MONGO DB

DB NoSql

Db non relazionali, senza relazioni tra tabelle ne vincoli (dati non strutturati); i db NoSql possono gestire dati in qualunque formato che possono essere memorizzati in modo flessibile e scalabiel, senza definire uno schema rigido a priori. Inoltre, i db NoSql sono progettati per scalare orizzontalmente su più server.

I db NoSql utilizzano vari modi:

- con sistema **chiave-valore**;
- con i **documenti**;
- con **modello a grafo**, con nodi ed archi;
- **modelli a colonne**, al posto che a righe;

MongoDB, NoSql orientato ai documenti; gestisce dati in **JSON**, organizzati in **collezioni** che rappresentano **gruppi di documenti correlati.**

MongoDB è scalabile orizzontalmente su più server tramite una tecnica chiamata **sharding,** che distribuisce i dati su più server. Supporta inoltre **l'indicizzazione flessibile**, che permette di indicizzare i dati in base a qualsiasi campo dei documenti, tecnica particolarmente performante su grande quantità di dati.

Le query vengono scritte con un modello basato su documenti. MongoDB permette di offrire una funzionalità avanzate l'aggregazione che consente di combinare e analizzare i dati su più documenti.

Mongo DB

Come lo creo? Seguo le istruzioni su mongo.db/atlas; dopo aver creato il db, devo creare un **cluster** (scegliendo cloud provider, regione, tipo di server etc..) e poi aggiungere gli account utente.

Poi configuro le opzioni di rete, connetto il server al DB e aggiungo i dati all'interno del db.

Per connettersi da remoto può essere usata la **MongoDB shell.** Alcuni dei suoi comandi:

- **show dbs:** visualizza lista db;
- **use <nome_db>:** seleziona il db su cui lavorare;
- **db.createCollection(<nome_collezione>):** creazione di una nuova collezione in un database specifico;
- **db.<nome_collezione>.drop():** eliminazione di una collezione esistente in un db specifico;

- **show collections:** visualizza lista delle collections;
- **db.<nome_collezione>.find():** visualizza tutti i documenti nella collezione specificata;
- **db.<nome_collezione>.insertOne(<documento>):** aggiunge un nuovo documento alla collezione specificata;
- **db.<nome_collezione>.insertMany(<documenti>)**
- **db.<nome_collezione>.updateOne(<filtro>, <aggiornamento>):** aggiorna il primo documento che soddisfa il filtro specificato;
- **db.<nome_collezione>.updateMany(<filtro>, <aggiornamento>):** aggiorna tutti i documenti che soddisfano il filtro specificato;
- **db.<nome_collezione>.deleteOne(<filtro>):** elimina il primo doc che risponde al filtro;
- **db.<nome_collezione>.deleteMany(<filtro>);**
- **db.<nome_collezione>.drop();** elimina la collezione specificata.

La collezione equivale sostanzialmente ad una tabella di un DB relazionale, ed ogni documento di essa rappresenta un record. Le collezioni sono **schemaless**, cioè ogni documento all'interno della collection può avere campi diversi. In MongoDB è possibile eseguire query utilizzando il **MongoDB Query Language (MQL) e il MongoDB Aggregation Framework.**

Esempio di comandi:

```
use mydb
db.createCollection("users")
db.users.insertMany([
  {name:"Paolo Bianchi",email: "paolo.bianchi@example.com"},
  {name:"Giulia Verdi",email: "giulia.verdi@example.com"},
  {name:"Luca Neri",email: "luca.neri@example.com"}
])
db.users.find()
db.users.insertOne(
  {name:"Mario Red",email:"mario.red@ex.com"})
db.users.find()
db.users.drop()
```

Per collegare una app Node.js ad un MongoDB, devo:

- autorizzare l'ip del server
- configurare la stringa di connessione, i.e.

  ```
  mongodb+srv://<username>:<password>@mydb.twv6equ
  .mongodb.net/<collection>
  ```

- Usare mongoose per configurare la nostra app node.js (utilizzando, ad esempio, una funzione di connessione asincrona):

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

async function connect() {
  try {
    await
mongoose.connect('mongodb+srv://<username>:<pwd>@mydb.twv6equ
.mongodb.net/<collection>', {
      useNewUrlParser: true,
      useUnifiedTopology: true
    });
    console.log('Connected to MongoDB');
  } catch (error) {
    console.error(error);
  }
}
```

- Esempio di funzione per la restituzione della prima collection:

```
async function getFirstCollection() {
    try {
      const collections = await
        mongoose.connection.db.listCollections().toArray();
      const firstCollection = collections[0];
      console.log(firstCollection.name)
      const Model =
        mongoose.model(
        firstCollection.name,
        new mongoose.Schema({}),firstCollection.name);
      const documents = await Model.find({});
      console.log(documents);
    } catch (error) {
      console.error(error);
    } finally {
      mongoose.disconnect();
    }
}
```

- Esempio di funzione per la ricerca di un documento:

```
async function searchDocument() {
    try {
      const Model = mongoose.model(
        'User', new mongoose.Schema({name: String}),'users');
      const document = await
        Model.findOne({name:'Paolo Bianchi'});
      console.log(document);
    } catch (error) {
      console.error(error);
    } finally {
      mongoose.disconnect();
    }
}
```

- **NB potrei utilizzare anche una sintassi con le promises:**

```
connect()
  //.then(() => listCollections())
  //.then(() => getFirstCollection())
  //.then(() => searchDocument())
  .then(() => searchDocuments())
  .catch((error) => console.error(error));
```

La sintassi di mongoose è quindi:

mongoose.model(name, schema, collection, skipInit) dove:

- **name** è il nome del modello;
- **schema** è lo schema Mongoose utilizzato per definire la struttura dei documenti della collezione;
- **collection** è il nome della collezione associata al modello del DB;;
- **skipInit** è un parametro bool facoltativo che indica se eseguire o meno l'inizializzazione automatica del modello (default: false);

Axios e Vue.js

Axios

libreria js per semplificare la gestione delle richieste http asincrone verso server; si basa su **promises**, quindi permette di gestire anche risorse sincrone.

Proviamo a considerare una richiesta GET all'indirizzo https://jsonplaceholder.typicode.com/posts per ricevere un array di oggetti json contenente informazioni sui post di un blog fittizio.

Consideriamo la chiamata tramite **Fetch API** (nativa per il browser, più macchinosa per la gestione di errori ed eccezioni):

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(response => response.json())
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.log(error);
  });
```

Considero la stessa chiamata in **Ajax JQuery** (lib esterna, sintassi semplice, personalizzazione delle richieste):

```
$.ajax({
  url: 'https://jsonplaceholder.typicode.com/posts',
  method: 'GET',
  success: function(response) {
    console.log(response);
  },
  error: function(error) {
    console.log(error);
  }
});
```

Se volessi utilizzare Axios invece (lib esterna, sintassi e gestione chiamate più semplice di jQuery)**:**

```
const axios = require('axios');
axios.get('https://jsonplaceholder.typicode.com/posts')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.log(error);
  });
```

Vue.js

Vue.js è un framework js opensource per la creazione di interfacce utente reattive e dinamiche; utilizza un pattern architetturale **Modello-Vista-ViewModel (MVVM),** che divide la logica del business dalla logica di presentazione.

Sistema di moduli e components, programmazione reattiva e logica dichiarativa.

ViewModel è una variante MVC che si concentra sulla separazione tra vista e modello. In questo pattern la vista rappresenta ancora l'interfaccia utente **ma la logica di presentazione viene spostata nel ViewModel, che si occupa di mappare i dati del modello alla vista e viceversa,** rendendola indipendente dalla logica di business (in MVC, è il controller a gestire la logica di business dell'applicazione).

Esempio di MVC in Vue.js per la gestione di un contatore:

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');
const CounterModel = require('./counterModel');
app.use(bodyParser.urlencoded({ extended: false }));

app.get('/', function (req, res) {
  const model = new CounterModel();
  res.render('view', { counter: model.getCounter() });
});

app.post('/', function (req, res) {
  const model = new CounterModel();
  if (req.body.increment) {model.incrementCounter();
  } else if (req.body.reset) {model.resetCounter();}
  res.render('view', { counter: model.getCounter() });
});

app.listen(3000, function () {
  console.log('App listening on port 3000');
});
```

MVC – index.js

```
class CounterModel {

  constructor() {this.counter = 0;}

  getCounter() {return this.counter;}

  incrementCounter() {this.counter++;}

  resetCounter() {this.counter = 0;}
}

module.exports = CounterModel;
```

MVC – counterModel.js

```
<html>
  <head>
    <title>MVC Example</title>
  </head>
  <body>
    <h1>MVC Example</h1>
    <p>Counter: <%= counter %></p>
    <form method="POST">
      <button name="increment">Increment</button>
      <button name="reset">Reset</button>
    </form>
  </body>
</html>
```

MVC – view.ejs

La stessa cosa con MVVM:

```
<template>
  <div>
    <h1>{{ title }}</h1>
    <p>{{ message }}</p>
    <button @click="incrementCounter()">Incrementa</button>
    <p>Contatore: {{ counter }}</p>
  </div>
</template>
<script>
export default {
  data() {
    return {
      title: 'Benvenuti in Vue.js!',
      message: 'Questo è un esempio di componente Vue.js.',
      counter: 0
    }
  },
  methods: {incrementCounter() {this.counter++}}
}
</script>
```

MVVM – index.js

Come possibile vedere, in **MVVM** viene creato un componente Vue.jss che visualizza un titolo, un messaggio, un pulsante ed un contatore; **la sezione template contiene il markup HTML del componente, mentre la sezione script contiene la logica Js che lo gestisce.**

In **MVC** invece, **il controller è responsabile di gestire la logica di business dell'app, compreso l'aggiornamento del contatore.**

AXIOS e VUE.JS vengono utilizzati insieme per la creazione di app web moderne e dinamiche.

Il componente **mounted()** è un metodo del ciclo di vita di un componente Vue.js che **viene chiamato quando un componente viene montato nel DOM, quando il componente è stato inizializzato ed il suo template viene renderizzato nella pagina.** Esso viene utilizzato una sola volta; se devo aggiornare dei dati, devo usare altri metodi quali **updated().**

Angular.JS

Framework per la creazione di applicazioni web; utilizza un pattern **MVC.** è scritto in TypeScript, che comprende la tipizzazione statica e la gestione delle classi. Viene distribuita tramite libreria e CDN.

Caratteristiche:

- **Modularità;**
- **Binding dei dati:** consente binding bidirezionale, collegando i dati del model alla View e di aggiornare automaticamente i dati quando vengono modificati;
- **Routing;**
- **Testing;** offre numerosi strumenti e librerie per testare l'applicazione;
- **Animazioni ed effetti visivi avanzati;**

modulo – controller - direttiva

Modulo: contenitore che raggruppa gli elementi dell'applicazione Angular (controller, direttive, filtri...). Si definisce con la funzione:

var app = angular.module('nomemodulo, []);

La quadra vuota indica che non ha dipendenze esterne.

Controller: funzione JS che gestisce il **modello** dell'app e fornisce i dati e le funzionalità necessarie alla vista. Si definisce come una funzione che accetta il parametro **$scope**, che rappresenta il contesto di esecuzione del controller.

app.controller('nomecontroller', function($scope){

$scope.name = "Mondo";

});

Angular JS usa la sintassi delle doppie graffe {{}} per l'interpolazione delle stringhe; questa sintassi permette di visualizzare il valore di una proprietà del controller all'interno del contenuto HTML.

Direttiva: etichetta HTML che definisce il comportamento di una vista e permette di estendere le funzionalità degli elementi HTML. Utilizza la funzione app.directive():

app.directive('nomedirettiva', function() {

return {

template: " Hello World! "

};

});

A quel punto, posso utilizzare l'etichetta <nomedirettiva></nomedirettiva> per generare automaticamente un paragrafo con la scritta "Hello World!" all'interno della vista.

Esistono direttive predefinite in AngularJS che si possono utilizzare al bisogno, tipo:

- **ng-app:** specifica il punto di partenza di una app JS, viene utilizzata come attributo sull'elemento HTML radice della pagina.
- **ng-init**: consente di inizializzare il valore di una variabile quando si carica la pagina HTML.
- **ng-controller:** specifica il controller da utilizzare per una sezione specifica della pagina HTML.
- **ng-model**: collega l'input dell'utente all'app Angular JS in maniera bidirezionale.
- **ng-bind**: collega un elemento HTML ad una proprietà del controller, in modo che il valore della proprietà venga visualizzato nell'elemento HTML (i.e. o) in maniera unidirezionale.
- **ng-repeat**: itera su un array o su un oggetto e crea una copia delle sezione HTML per ogni elemento nell'array o nell'oggetto.

Lo scope, come già detto, è l'oggetto JS che rappresenta il modello dell'appllicazione in Angular: contiene le proprietà e i metodi che sono disponibili per la vista ed il controller. Lo scope è, in sostanza, il tramite tra vista e modello.

```
<div ng-app="myApp" ng-controller="myCtrl">
<h1>{{carname}}</h1>
</div>

<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.carname = "Ferrari";
});
</script>
```

In questo esempio di codice, se abbiamo definito la proprietà $scope.carname nel controller, possiamo utilizzare la sintassi di interpolazione {{carname}} oppure la direttiva ng-bind=”carname” per visualizzare il valore della proprietà.

Ovviamente possiamo avere anche più controller con compiti diversi:

```
<div ng-controller="myCtrl">
    <h1>{{ greeting }}</h1>
      <input type="text" ng-model="name">
    <p>Hello, {{ name }}!</p>
  </div>
  <div ng-controller="userController">
    <h1>Users</h1>
    <ul>
      <li ng-repeat="user in users">
        {{ user.name }} ({{ user.email }})
      </li>
    </ul>
  </div>
```

Nel seguente contesto, si utilizza invece ng-init per inizializzare la variabile $scope.firstname con il valore “John”; **la direttiva ng-bind viene utilizzata per legare il valore della variabile $scope.fisrtName all’elemento HTML .**

```
<div ng-app="" ng-init="firstName='John'">
<p>The name is <span ng-bind="firstName"></span></p>
</div>
```

Eventi

Rappresentano l’interazione tra l’utente e l’interfaccia. Angular fornisce direttive dedicate per la gestione degli eventi, tra cui listener e callback.

Alcuni esempi:

- ng-click
- ng-mousedown
- ng-copy

- ng-keydown
- etc...

Esempio di utilizzo:

```
<button ng-click="showMessage()">Cliccami!</button>


app.controller('MyController', function($scope) {
  $scope.message = "Ciao mondo!";

  $scope.showMessage = function() {
    alert($scope.message);
  };
});
```

Forms

Possiamo utilizzare ng-model per associare un campo di input ad una variabile del controller; angular fornisce inoltre molte direttive per la validazione dei campi di input come ng-required, ng-pattern, ng-minlength etc...

Angular fornisce anche la direttiva ng-reset per reimpostare tutti i campi del form.