# 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).