Domande generiche

Le risposte sono:

1. **Cos'è CORS?** CORS, acronimo di Cross-Origin Resource Sharing, è un meccanismo di sicurezza implementato nei browser che permette di controllare quali risorse possono essere richieste da un dominio diverso rispetto a quello da cui la risorsa è stata servita. Viene utilizzato per prevenire richieste HTTP non autorizzate da domini esterni, aumentando la sicurezza delle applicazioni web.
2. **Metodi HTTP** I metodi HTTP sono verbi che indicano l'azione da eseguire su una risorsa specifica. I metodi più comuni sono:
    - GET: recupera una risorsa.
    - POST: invia dati al server per creare una nuova risorsa.
    - PUT: aggiorna una risorsa esistente.
    - DELETE: elimina una risorsa.
    - HEAD: recupera le intestazioni di una risorsa.
    - OPTIONS: descrive le opzioni di comunicazione per la risorsa.
    - PATCH: applica modifiche parziali a una risorsa.
3. **Cosa si intende per semantica degli elementi HTTP?** La semantica degli elementi HTTP si riferisce al significato e allo scopo specifico di ciascun metodo, intestazione, e codice di stato HTTP. Ad esempio, un metodo GET semantizza la richiesta di lettura di una risorsa senza effetti collaterali, mentre un POST indica l'invio di dati al server con potenziale creazione o modifica di risorse.
4. **Parlami delle API, cosa fai dal punto di vista del server per fare richieste agli altri siti?** Le API (Application Programming Interface) sono interfacce che permettono a diverse applicazioni software di comunicare tra loro. Dal punto di vista del server, per fare richieste ad altri siti, si utilizza un client HTTP, che può essere una libreria come axios o fetch in JavaScript, per inviare richieste e ricevere risposte da API esterne. Il server può fare richieste HTTP come GET, POST, PUT, DELETE, ecc., e deve gestire le risposte, incluso il trattamento di eventuali errori.
5. **Serif e sans Serif** I caratteri serif sono caratterizzati dalla presenza di piccole linee o abbellimenti (serif) alla fine delle lettere. I caratteri sans serif, al contrario, sono privi di queste linee aggiuntive. Esempi di font serif includono Times New Roman e Georgia, mentre Arial e Helvetica sono esempi di sans serif.
6. **Sapresti fare un middleware, che callback gli passi?** Un middleware è una funzione che ha accesso all'oggetto request, response e alla funzione next nell'applicazione Express.js. Un esempio di middleware potrebbe essere:
7. function logRequest(req, res, next) {
8. console.log(`${req.method} ${req.url}`);
9. next();
10. }

app.use(logRequest);

La callback next viene chiamata per passare il controllo al middleware successivo.

11. **Caratteristiche di JavaScript** JavaScript è un linguaggio di programmazione dinamico e interpretato, noto per le seguenti caratteristiche:
    - Basato sugli eventi e orientato agli oggetti.
    - Supporta funzioni di prima classe.
    - Ha un'ampia compatibilità con i browser.
    - Utilizza la gestione asincrona tramite callback, Promises e async/await.
    - Permette la manipolazione del DOM (Document Object Model).
12. **La specificità del CSS** La specificità del CSS è una misura che determina quali regole CSS verranno applicate a un elemento specifico quando ci sono regole contrastanti. È calcolata basandosi sul numero di selettori di tipo ID, classi e pseudo-classi, e selettori di tipo presenti in un selettore. Più specifica è una regola, maggiore è la priorità che ha.
13. **Cosa sono i web font?** I web font sono caratteri tipografici che possono essere caricati e utilizzati nelle pagine web tramite CSS. Permettono agli sviluppatori di utilizzare font non standardizzati o personalizzati che non sono installati localmente sul dispositivo dell'utente.
14. **Font-Family** La proprietà font-family in CSS specifica una lista di famiglie di caratteri per un elemento. La lista può includere font generici come serif, sans-serif, monospace, ecc., e font specifici come Arial, Helvetica, Times New Roman, ecc. Esempio:

```
p {
  font-family: "Arial", sans-serif;
}
```

11. **Cos'è il font stack?** Il font stack è una lista di famiglie di caratteri specificata nella proprietà font-family. Questa lista permette al browser di utilizzare il primo font disponibile nell'elenco. Se il primo non è disponibile, verrà utilizzato il successivo, e così via.
12. **Desktop first e mobile first** Desktop first e mobile first sono due approcci alla progettazione responsive:

- Desktop first: il design della pagina web viene creato inizialmente per i dispositivi desktop e poi adattato ai dispositivi mobili utilizzando media queries. (max-width)
- Mobile first: il design viene inizialmente creato per dispositivi mobili e successivamente adattato per dispositivi desktop con media queries. (min-width)

13. **Cos'è il collasso dei margini?** Il collasso dei margini si verifica quando i margini verticali di due elementi adiacenti (superiore e inferiore) si combinano in un unico margine. Il margine risultante sarà pari al maggiore dei due margini coinvolti.
14. **Architettura di Node.js, com'è fatta, quanti thread ha, ha le code?** vedi anche qui Node.js è basato su un'architettura event-driven e non-bloccante. Utilizza un singolo thread principale per l'esecuzione del codice JavaScript, ma sfrutta un pool di thread del sistema operativo per

operazioni di I/O asincrone. Utilizza anche una coda di eventi per gestire le operazioni asincrone.

15. **Differenza tra margin e padding**

- Margin: è lo spazio esterno che separa un elemento dagli altri elementi circostanti.
- Padding: è lo spazio interno tra il contenuto di un elemento e il suo bordo.

16. **Border-box** La proprietà box-sizing: border-box in CSS modifica il comportamento del modello di box. Quando è impostata, la larghezza e l'altezza di un elemento includono il padding e il bordo, ma non il margine.

17. **Head e Header**

- <head>: un elemento HTML che contiene metadati per il documento, come il titolo, i link ai fogli di stile, e i metadati.
- <header>: un elemento HTML che rappresenta un contenitore per contenuti introduttivi o di navigazione, come titoli, loghi, e menu di navigazione.

18. **CSS Position: static, fixed, relative, absolute**

- static: è il valore predefinito, l'elemento è posizionato secondo il normale flusso del documento.
- fixed: l'elemento è posizionato rispetto alla finestra del browser e non si muove quando la pagina viene scrollata.
- relative: l'elemento è posizionato rispetto alla sua posizione originale nel flusso del documento.
- absolute: l'elemento è posizionato rispetto all'elemento contenitore più vicino che ha una posizione diversa da static.

19. **Media query** Le media queries sono una caratteristica del CSS3 che permette di applicare stili differenti a seconda delle caratteristiche del dispositivo, come la larghezza della finestra del browser, la risoluzione dello schermo, e l'orientamento. Esempio:

```
@media (max-width: 600px) {
  body {
    background-color: lightblue;
  }
}
```

20. **Tecniche responsive** Le tecniche responsive includono:

- Uso di layout fluidi basati su percentuali piuttosto che su pixel.
- Utilizzo di media queries per adattare il design ai diversi dispositivi.
- Implementazione di immagini responsive che si ridimensionano in base alla dimensione dello schermo.
- Applicazione di griglie flessibili e framework CSS come Bootstrap o Foundation.

21. **FORM** Un form in HTML è un contenitore per elementi di input che permette agli utenti di inviare dati a un server. Può includere campi di testo, caselle di controllo, pulsanti radio, menu a discesa e pulsanti di invio. Esempio:

```
<form action="/submit" method="post">
  <label for="name">Nome:</label>
  <input type="text" id="name" name="name">
  <input type="submit" value="Invia">
</form>
```

22. **Chi è il this nel lato function?** In una funzione, il valore di this dipende da come la funzione è chiamata. Quando una funzione è chiamata come metodo di un oggetto, this si riferisce all'oggetto stesso. Quando una funzione è chiamata come funzione standalone, this si riferisce all'oggetto globale (window in browser, global in Node.js), o undefined in strict mode.
23. **Cos'è una promise?** Una Promise è un oggetto che rappresenta l'eventuale completamento o fallimento di un'operazione asincrona. Ha tre stati: pending (in attesa), fulfilled (completata con successo), e rejected (fallita).
24. **Come creare una promise?** Una promise può essere creata utilizzando il costruttore Promise, che prende una funzione executor con due parametri: resolve e reject. Esempio:

```
let myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Operazione completata");
  }, 1000);
});
```

25. **Cos'è la coda di microtask?** La coda di microtask è una coda di operazioni che devono essere eseguite subito dopo il completamento dell'attuale task in corso. Le promesse risolte e le callback di MutationObserver sono esempi di microtask.
26. **Transpilers** I transpilers sono strumenti che traducono il codice sorgente scritto in un linguaggio di programmazione in un altro linguaggio di programmazione. Ad esempio, Babel è un transpiler che converte il codice ES6+ in ES5 per compatibilità con i browser più vecchi.
27. **Polyfill** Un polyfill è un pezzo di codice (di solito JavaScript) che implementa funzionalità che non sono nativamente supportate in alcuni browser. Viene utilizzato per garantire che una funzionalità moderna sia disponibile in tutti gli ambienti di esecuzione.
28. **Garbage collector** Il garbage collector è un sistema automatizzato di gestione della memoria che recupera memoria inutilizzata liberando gli oggetti che non sono più raggiungibili o utilizzati dal programma.
29. **Scope** Lo scope (ambito) definisce la visibilità e la durata delle variabili e delle funzioni. In JavaScript, ci sono due tipi principali di scope: lo scope globale e lo scope locale (o di funzione). Con l'introduzione di let e const, esiste anche lo scope di blocco. Lo **scope locale** lo hanno quelle variabili che si trovano all'interno delle funzioni, mentre quando una variabile si trova fuori dalla funzione assume **scope globale** Esempio :

```
let var1; //scope globale


function myFunction(){
        let var2; //scope locale
        // Do Stuff...
}
console.log(var1) //accedo a var1 perchè ha scope globale
console.log(var2) //non riesco perchè var2 è definita nella funzione
```

30. **Costruttore** Un costruttore è una funzione speciale in una classe o un oggetto che viene chiamata per inizializzare l'oggetto appena creato. In JavaScript, le classi possono avere un metodo constructor per questo scopo. Quando viene chiamato un costruttore con new:

- Viene creato un oggetto vuoto e assegnato a this
- Viene eseguita la funzione
- Viene ritornato this Lo scope di **var** è il functional block più vicino. Lo scope di **let** è l'enclosing block più vicino

1. **DOM** Il DOM (Document Object Model) è una rappresentazione strutturata di un documento HTML o XML sotto forma di un albero. Permette a linguaggi di programmazione come JavaScript di manipolare il contenuto, la struttura e lo stile di un documento web.
2. **Codice sincrono/asincrono**

- Codice sincrono: viene eseguito in sequenza, ogni istruzione attende la fine della precedente.
- Codice asincrono: permette l'esecuzione di operazioni senza bloccare il flusso principale del programma, utilizzando callback, Promises, o async/await.

33. **Microtask** vedi anche qui

I microtask sono operazioni che devono essere eseguite immediatamente dopo il completamento dell'attuale task in corso. Esempi di microtask includono la risoluzione delle promesse e le callback dei MutationObserver.

34. **Architettura REST, con principi HTTP** REST (Representational State Transfer) è un'architettura per la progettazione di API basata su principi HTTP come:

- Risorse identificate da URL.
- Uso di metodi HTTP standard (GET, POST, PUT, DELETE).
- Operazioni stateless.
- Uso di rappresentazioni multiple (XML, JSON) per risorse.

35. **Come fare un'API REST** Per creare un'API REST, seguire questi passaggi:

- Definire le risorse e i relativi endpoint.
- Implementare metodi HTTP per ogni endpoint.

- Gestire le richieste e risposte, assicurandosi di utilizzare codici di stato HTTP appropriati.
- Assicurare la statelessness, mantenendo il contesto delle richieste sul client.
- Implementare la gestione degli errori e la sicurezza (ad esempio, autenticazione e autorizzazione).

36. **Cosa sono le API?** Le API (Application Programming Interface) sono insiemi di definizioni e protocolli che permettono a diverse applicazioni software di comunicare tra loro. Le API specificano come le diverse componenti software devono interagire, definendo i metodi e i dati che possono essere utilizzati per effettuare richieste e ricevere risposte.

Event Loop di JavaScript

Definizione

L'event loop è un costrutto di programmazione che gestisce le operazioni asincrone, consentendo a JavaScript di eseguire il codice, raccogliere e elaborare eventi e sottoscrivere operazioni eseguite, il tutto senza bloccare il thread principale.

Funzionamento

1. **Stack di Chiamate (Call Stack)**: JavaScript è single-threaded, il che significa che ha un unico stack di chiamate in cui vengono eseguite le funzioni. Ogni funzione invocata viene aggiunta a questo stack, e una volta terminata, viene rimossa.
2. **Coda dei Messaggi (Message Queue)**: È una struttura dati in cui vengono messi i messaggi contenenti funzioni callback da eseguire. Questi messaggi sono in attesa di essere elaborati una volta che lo stack di chiamate è vuoto.
3. **Event Loop**: È il meccanismo che controlla lo stack di chiamate e la coda dei messaggi. Quando lo stack di chiamate è vuoto, l'event loop prende la prima funzione dalla coda dei messaggi e la mette nello stack di chiamate per l'esecuzione. Questo processo continua ciclicamente.

Node.js

^150eed

Definizione

Node.js è un runtime JavaScript costruito sul motore V8 di Google Chrome, che permette di eseguire JavaScript lato server. Node.js è noto per il suo modello di I/O non bloccante, orientato agli eventi, che lo rende ideale per applicazioni che necessitano di alte prestazioni e scalabilità, come i server web.

Funzionamento

Node.js utilizza l'event loop per gestire le operazioni di I/O asincrone. Invece di bloccare il thread principale durante operazioni I/O, come la lettura da file o l'invio di richieste di rete, Node.js delega queste operazioni a un pool di thread worker (fornito dalla libreria libuv), che gestisce le operazioni I/O in background. Quando un'operazione I/O è completata, una callback corrispondente viene messa nella coda dei messaggi, e l'event loop si occupa di eseguirla non appena lo stack di chiamate è vuoto.

Architettura di Node.js

Architettura Event-Driven e Non-Blocking I/O

1. **Single Thread**: Node.js utilizza un singolo thread per eseguire il codice JavaScript.
2. **Libuv**: Una libreria multi-piattaforma che gestisce il pool di thread, la gestione degli eventi e le operazioni I/O asincrone.
3. **Event Loop**: Come descritto sopra, il cuore del modello di concorrenza di Node.js.
4. **Callback**: Funzioni che vengono passate come argomenti ad altre funzioni per essere eseguite dopo il completamento di operazioni asincrone.
5. **EventEmitter**: Un modulo che facilita la gestione degli eventi e delle callback.

Vantaggi

- **Alta Concorrenza**: L'architettura non bloccante permette a Node.js di gestire un elevato numero di connessioni contemporanee senza bloccare il thread principale.
- **Efficienza**: La delega delle operazioni I/O a thread worker permette di liberare il thread principale per altre operazioni, migliorando l'efficienza complessiva.

Conclusione

L'event loop è il meccanismo che permette a JavaScript di gestire le operazioni asincrone in modo efficiente, mentre Node.js sfrutta questo meccanismo insieme a un'architettura event-driven e non-blocking I/O per gestire operazioni server-side ad alta concorrenza e scalabilità. Queste caratteristiche rendono Node.js particolarmente adatto per applicazioni web in tempo reale, come server HTTP e applicazioni basate su WebSocket.

Microtask

^19a2f8

Le microtask sono una parte fondamentale dell'architettura di gestione degli eventi e dell'esecuzione asincrona in JavaScript. Sono un tipo di attività che viene eseguita immediatamente dopo che l'event loop ha completato il ciclo corrente, ma prima che riprenda a elaborare la coda dei messaggi (macro-task).

Microtask nel Contesto dello Sviluppo Web

Definizione

Le microtask sono piccole unità di lavoro che vengono eseguite subito dopo la conclusione della funzione corrente e prima che il ciclo dell'event loop venga completato. Vengono spesso utilizzate per operazioni che devono essere eseguite rapidamente e che hanno una priorità maggiore rispetto alle macro-task (come eventi I/O, timeout, ecc.).

Esempi di Microtask

- **Promesse (Promises)**: Le callback .then e .catch di una promessa vengono eseguite come microtask.
- **Mutation Observer**: Callback utilizzate per osservare e reagire alle modifiche nel DOM.

Funzionamento

Quando un'operazione asincrona, come una promessa, viene risolta, il suo callback viene messo nella coda delle microtask. Dopo che lo stack di chiamate corrente è vuoto, l'event loop controlla la

coda delle microtask e esegue tutte le microtask in quella coda prima di passare alle macro-task (i normali eventi asincroni, come setTimeout e setInterval).

Priorità delle Microtask

Le microtask hanno una priorità maggiore rispetto alle macro-task. Questo significa che se ci sono microtask nella coda, queste verranno eseguite prima che venga eseguita qualsiasi macro-task successiva. Questo garantisce che le operazioni che necessitano di una rapida risposta vengano gestite prontamente.

Esempio di Codice

Ecco un esempio per illustrare la differenza tra microtask e macro-task:

```
console.log('Start');


setTimeout(() => {
  console.log('setTimeout');
}, 0);


Promise.resolve().then(() => {
  console.log('Promise');
});


console.log('End');
```

- **Start** e **End** vengono loggati immediatamente perché sono operazioni sincrone.
- **Promise** viene loggato prima di **setTimeout** perché la callback della promessa è una microtask, mentre setTimeout è una macro-task.

Differenza tra Microtask e Macro-task

- **Microtask**: Hanno priorità più alta e vengono eseguite subito dopo che lo stack di chiamate è vuoto, prima che l'event loop inizi un nuovo ciclo.
    - Esempi: Promises, Mutation Observers.
- **Macro-task**: Vengono eseguite dopo che tutte le microtask sono state elaborate.
    - Esempi: setTimeout, setInterval, I/O events, rendering.

Utilità delle Microtask

- **Reattività**: Le microtask permettono una gestione più reattiva delle operazioni asincrone, migliorando l'esperienza utente nelle applicazioni web.
- **Consistenza**: Garantiscono che le operazioni importanti vengano completate prima che l'event loop continui con altre operazioni di minor priorità.

In conclusione, le microtask sono essenziali per la gestione efficiente delle operazioni asincrone ad alta priorità in JavaScript, garantendo che le promesse e altre operazioni critiche vengano gestite rapidamente e in modo ordinato.

Routing

Il **routing** nello sviluppo web si riferisce al processo di determinazione di quale azione intraprendere in risposta a una richiesta di un URL specifico. È una parte fondamentale del funzionamento di applicazioni web, server web e framework lato client.

Routing lato Server

Nelle applicazioni web lato server, il routing è il meccanismo che associa gli URL alle funzioni del server o ai controller che gestiscono le richieste. Quando un server web riceve una richiesta, il router analizza l'URL e decide quale codice eseguire per rispondere alla richiesta.

Esempio con Express.js (Node.js)

In un'applicazione Express, il routing è configurato definendo le route e le relative funzioni di gestione:

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

const app = express();


// Definisce una route per la home page

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

  res.send('Benvenuto alla home page!');

});


// Definisce una route per una pagina di contatto

app.get('/contatti', (req, res) => {

  res.send('Questa è la pagina dei contatti.');

});


// Definisce una route per gestire parametri dinamici

app.get('/utente/:id', (req, res) => {

  const userId = req.params.id;

  res.send(` Profilo utente con ID: ${userId}` );

});


app.listen(3000, () => {

  console.log('Server in ascolto su http://localhost:3000');
```

});

In questo esempio:

- app.get('/'): Gestisce le richieste alla home page.
- app.get('/contatti'): Gestisce le richieste alla pagina dei contatti.
- app.get('/utente/:id'): Gestisce le richieste a una pagina utente con un ID dinamico.

Tipologie di Routing

1. **Routing Statico**: Gli URL sono mappati a risorse specifiche, come file o funzioni. È rigido e non supporta parametri dinamici.
2. **Routing Dinamico**: Gli URL possono includere parametri dinamici, come /:id, che vengono gestiti in modo flessibile.
3. **Routing Nested**: Permette di definire rotte annidate, utile per applicazioni complesse con sezioni diverse che richiedono sub-routes.

Benefici del Routing

- **Organizzazione del codice**: Separa le logiche di gestione delle richieste, migliorando la manutenibilità del codice.
- **Navigazione user-friendly**: Permette una navigazione intuitiva con URL significativi.
- **Sviluppo di SPA**: Facilita il caricamento dinamico dei contenuti senza ricaricare l'intera pagina.

Conclusione

Il routing è un concetto cruciale nello sviluppo web, sia lato server che lato client. Permette di associare URL specifici a funzioni di gestione o componenti, migliorando l'organizzazione del codice e l'esperienza utente.