1. Sommario

:+1: Premere sulle frecce per visualizzare il codice :+1:V

2. Introduzione alla programmazione

2.1. Best Practice di programmazione

In generale, la scelta di usare const dove possibile è considerata una buona pratica di programmazione in JavaScript moderno. Ti aiuta a scrivere codice più sicuro e mantenibile. L'utilizzo di `const` invece di `let` in JavaScript ha diversi vantaggi che possono migliorare la qualità e l'affidabilità del codice:

1. **Immutabilità**: Una volta assegnato un valore a una variabile dichiarata con const, non può essere riassegnato. Questo aiuta a prevenire errori in cui potresti accidentalmente cambiare il valore di una variabile. È importante notare che const non rende l'oggetto a cui si riferisce immutabile, solo la riassegnazione della variabile è proibita.

2. **Leggibilità e Intenzione**: Quando usi const, comunichi agli altri sviluppatori che stai lavorando con che il valore della variabile non dovrebbe cambiare. Questo rende il codice più leggibile e chiaro. Chi legge il codice può essere sicuro che il valore associato a quella variabile rimarrà lo stesso durante tutto il suo `ambito (scope)`.

3. **Prevenzione di Errori**: Aiuta a prevenire errori causati dalla riassegnazione accidentale di variabili, il che può essere particolarmente utile in progetti più grandi dove tracciare ogni variabile può essere difficile.

4. **Ottimizzazione del Motore JavaScript**: I motori JavaScript possono ottimizzare meglio il codice quando sanno che certi valori non cambieranno. L'uso di const può potenzialmente portare a prestazioni leggermente migliori, anche se questo vantaggio può essere minimo nella pratica.

5. **Uso del Blocco di Ambito**: Sia `let` che `const` sono variabili di `blocco (block-scoped)`, il che significa che esistono solo all'interno del blocco in cui sono stati dichiarati. Questo è più sicuro rispetto all'utilizzo di `var`, che è `funzione-scoped`.

6. **Buone Pratiche di Programmazione**: Usare `const` per default e `let` solo quando è necessario riassegnare il valore promuove una programmazione più intenzionale e precauzionale, riducendo la possibilità di effetti collaterali indesiderati nel codice.

2.2. Best practice per la gestione degli errori

Implementando queste best practice, puoi notevolmente migliorare la robustezza e l'affidabilità della tua applicazione, gestendo in modo efficace situazioni inaspettate e errori.

1. **Controllo del Tipo di Risposta**: Prima di analizzare la risposta come XML, verifica che il tipo di contenuto sia effettivamente XML. Questo può essere fatto controllando l'header `Content-Type` della risposta.

2. **Gestione Errori nel Parsing XML**: Quando usi DOMParser per analizzare la stringa XML, circonda il codice con un blocco `try...catch` per catturare eventuali errori di parsing.

3. **Verifica dello Stato della Risposta HTTP**: Prima di procedere con l'elaborazione della risposta, verifica che lo stato `HTTP sia 200`, indicando che la richiesta è stata un successo.

4. **Controllo dell'Esistenza dei Dati**: Prima di accedere ai nodi o agli attributi dell'XML, verifica che esistano effettivamente per evitare errori di tipo `null` o `undefined`.

5. **Gestione degli Errori della Richiesta**: Implementa una gestione degli errori per la richiesta `XMLHTTPRequest`. Usa l'evento onerror per catturare errori di rete e gestire situazioni in cui la richiesta non può essere completata.

6. **Messaggi di Errore Chiari**: Fornisci messaggi di errore chiari e specifici in modo che sia più facile capire cosa è andato storto. Questo è particolarmente utile durante lo sviluppo e il `debug`.

7. **Logging**: Usa il logging per registrare errori in modo che possano essere esaminati in seguito. Questo è utile per identificare problemi ricorrenti o per tenere traccia degli errori in produzione.

8. **Fallback e Recovery**: Implementa strategie di `fallback` o `recupero` in caso di errore. Ad esempio, potresti mostrare un messaggio all'utente o riprovare la richiesta dopo un breve ritardo.

9. **Feedback all'Utente**: In caso di errore, fornisci un `feedback` appropriato all'utente. Evita di mostrare errori tecnici direttamente, ma piuttosto comunica che si è verificato un problema in un linguaggio comprensibile.

10. **Validazione del Contenuto XML**: Assicurati che il contenuto dell'XML sia valido e sicuro prima di utilizzarlo. Questo è importante per prevenire attacchi come l'`XML External Entity (XXE)`.

2.3. Best practice per il refactoring

Il refactoring è il processo di migliorare e semplificare il codice senza cambiarne le funzionalità. In sintesi, il refactoring delle funzioni non solo rende il codice più pulito e gestibile, ma migliora anche l'efficienza generale del processo di sviluppo del software.

1. **Riduzione degli Errori**: Le funzioni più piccole e focalizzate tendono ad avere meno errori, poiché è più semplice verificare e testare la loro logica. Inoltre, se un errore si verifica in una funzione, è più facile isolarlo e correggerlo.

2. **Miglioramento delle Prestazioni**: In alcuni casi, il refactoring può portare a miglioramenti delle prestazioni, specialmente se permette di eliminare la duplicazione del codice o di ottimizzare determinate operazioni.

3. **Facilità di Testing**: Il codice organizzato in funzioni più piccole e ben definite è molto più facile da testare. Ogni funzione può essere testata individualmente, rendendo più semplice identificare e correggere i bug.

4. **Incapsulamento**: Il refactoring aiuta a incapsulare la logica in modo che ogni funzione gestisca una specifica operazione o calcolo. Questo isolamento della logica aiuta a proteggere il resto del programma da eventuali cambiamenti o problemi all'interno di una singola funzione.

5. **Semplificazione del Debugging**: Quando un errore si verifica in un programma con funzioni ben organizzate, è più facile tracciare la fonte del problema. Invece di dover esaminare un lungo pezzo di codice, puoi concentrarti su una funzione specifica.

6. **Miglioramento della Collaborazione**: In un ambiente di squadra, il refactoring in funzioni più piccole consente a più sviluppatori di lavorare contemporaneamente su parti diverse del codice senza interferire gli uni con gli altri.

7. **Pratiche di Programmazione Solide**: Infine, il refactoring è un segno di buone pratiche di programmazione. Dimostra un approccio ponderato allo sviluppo del software, dove la chiarezza, la manutenibilità e la qualità del codice sono prioritarie.

2.4. Best practice per eliminare la ridondanza

Nella tua funzione createTable, hai attualmente due loop separati per aggiungere le intestazioni della tabella: uno per iterare sui childNodes e l'altro per gli attributes di un nodo. Per rendere il codice più conciso e ridurre la ridondanza, puoi combinare questi due loop in uno. In questo codice, uso l'operatore di spread

- `[...nodes[0].childNodes, ...nodes[0].attributes]`

per creare un unico array che contiene sia `i childNodes` sia gli `attributes del primo nodo`. Questo array è quindi iterato una sola volta per creare le intestazioni della tabella, riducendo così la ridondanza del codice.

2.5. Analisi delle varie funzionalità del codice

Il codice in questione implementa una robusta soluzione per il parsing, il caricamento e la visualizzazione di dati XML in un contesto web, utilizzando JavaScript puro. La logica è strutturata per garantire una gestione efficace e un'interazione dinamica con i dati XML. Di seguito, vengono analizzate le principali caratteristiche tecniche e le scelte implementative:

2.5.1. Gestione del Ciclo di Vita della Richiesta XMLHTTPRequest:

Il codice inizia con window.onload, che assicura l'inizializzazione solo dopo il completo caricamento del DOM. Le funzioni `readXmlFile("XML/comuni.xml")` e `readXmlFile("XML/compositori.xml")` sono chiamate per avviare il processo di recupero dei dati. All'interno di `readXmlFile`, viene creato un oggetto `XMLHttpRequest`, e sono impostate le configurazioni per una richiesta `GET` asincrona `(xhr.open("GET", url, true))`. La natura asincrona di questa richiesta consente all'utente di continuare a interagire con la pagina mentre i dati vengono caricati.

2.5.2. Robusta Gestione degli Errori:

Gli errori di rete e i timeout sono gestiti rispettivamente da xhr.onerror e xhr.ontimeout, che chiamano `handleGlobalError`. Inoltre, `handleReadyStateChange` controlla lo stato della richiesta e il `codice di stato HTTP` per gestire le risposte non riuscite o non valide. Questo assicura che qualsiasi problema durante la richiesta venga catturato e gestito in modo appropriato.

2.5.3. Tentativi di Ripetizione della Richiesta:

La funzione `retryRequest` implementa una logica di ripetizione, chiamando ricorsivamente readXmlFile con un numero decrescente di tentativi e un ritardo (`setTimeout`). Questo approccio introduce una tolleranza ai guasti temporanei, come problemi di rete momentanei.

2.5.4. Validazione e Elaborazione dei Dati XML:

`processXMLResponse` si occupa dell'elaborazione iniziale della risposta, rimuovendo gli spazi bianchi inutili e convertendo il testo in un documento `XML (DOMParser)`. `validateXML` controlla poi se il documento XML è ben formato e se non contiene errori, lanciando un'eccezione in caso contrario.

2.5.5. Dinamicità nella Creazione del Contenuto del DOM:

`createTable` e `populateTable` utilizzano `DocumentFragment` per costruire la struttura della tabella in memoria prima di aggiungerla al DOM, `minimizzando il reflow e il repaint`. `createTable` costruisce l'header della tabella basandosi sui nodi e sugli attributi del primo elemento XML, mentre `populateTable` riempie il corpo della tabella con i dati.

2.5.6. Separazione e Modularità delle Funzionalità:

Il codice è organizzato in funzioni specifiche e autocontenute, ciascuna responsabile di un singolo aspetto del processo complessivo. Ad esempio, `updateProgressBar` gestisce esclusivamente l'aggiornamento della barra di progresso, mentre showErrorToUser si occupa solo di mostrare gli errori all'utente. Questa separazione chiara

rende il codice più leggibile, manutenibile e riutilizzabile.

2.5.7. Interazione con l'Utente:

Il codice mantiene l'utente informato su ciò che sta accadendo attraverso l'aggiornamento di una barra di progresso (`updateProgressBar`) e la visualizzazione di messaggi di errore (`showErrorToUser`). Ciò migliora l'esperienza dell'utente, fornendo feedback visivo e informazioni utili in caso di problemi.

2.5.8. Conclusioni

Questo approccio strutturato e dettagliato nel trattamento delle richieste asincrone, nella gestione degli errori e nell'interazione con il DOM e l'utente rende il codice un esempio solido di buone pratiche di programmazione in un contesto di applicazione web che utilizza dati XML.

2.6. Analisi delle singole funzioni che compongono il codice

2.6.1. window.onload

Funzionamento: Questa funzione viene invocata non appena il DOM della pagina è completamente caricato. È il punto di ingresso per l'esecuzione del codice, garantendo che tutti gli elementi della pagina siano accessibili.

Codice Javascript

```
window.onload = function () {
// Carica le tabelle
    try {
        readXmlFile("XML/comuni.xml"); // Leggi il file XML dei comuni
        readXmlFile("XML/compositori.xml"); // Leggi il file XML dei compositori
    } catch (error) {
        console.error("Errore nel caricamento dei file XML: ", error); // Gestisci eventuali errori nel caricamento dei file
    }
}
```

[!NOTE]

Questa funzione è un evento che si attiva quando tutto il contenuto della pagina è stato completamente caricato. All'interno di questa funzione, si possono vedere delle linee di codice commentate che servirebbero per inizializzare il processo di caricamento dei file XML. Tuttavia, sono state lasciate commentate per motivi dimostrativi.

2.6.2. readXmlFile(url)

Funzionamento: Inizializza e invia una richiesta XMLHTTP al percorso specificato dal parametro url. Configura vari gestori di eventi per la richiesta:

[!NOTE]

1. `onprogress`: Invoca updateProgress per aggiornare la barra di progresso durante il download del file.
2. `onreadystatechange`: Assegna handleReadyStateChange per gestire i cambiamenti di stato della richiesta, inclusa la validazione della risposta.
3. `onerror`: Invoca handleGlobalError in caso di errore di rete e tenta di ripetere la richiesta con retryRequest.
4. `ontimeout`: Gestisce i timeout della richiesta, segnalando l'errore tramite handleGlobalError.

Codice Javascript

```
function readXmlFile(url) {
    console.log("XMLHTTP request started for: ", url); // Log dell'inizio della richiesta
    const xhr = new XMLHttpRequest(); // Crea un nuovo oggetto XMLHttpRequest

    xhr.open("GET", url, true); // Configura la richiesta GET per l'URL specificato

    xhr.onprogress = updateProgress; // Assegna la funzione di aggiornamento della barra di progresso
    xhr.onreadystatechange = function (){handleReadyStateChange(xhr, url)}; // Gestisci i cambiamenti di stato della richiesta
    xhr.onerror = function() {
        handleGlobalError("Network error", url); // Gestisci errori di rete
        retryRequest(url, 3); // Riprova la richiesta in caso di errore
    };
    xhr.ontimeout = () => handleGlobalError("Timeout error", url); // Gestisci errori di timeout

    xhr.send(); // Invia la richiesta
```

```
}
```

2.6.3. updateProgress(event)

Funzionamento: Calcola e aggiorna la barra di progresso basandosi sulla quantità di dati già scaricati in proporzione al totale. Usa l'oggetto event per ottenere i valori necessari al calcolo della percentuale.

Codice Javascript

```
function updateProgress(event) {
    if (event.lengthComputable) {
        const percentComplete = (event.loaded / event.total) * 100; // Calcola la percentuale di completamento
        updateProgressBar(percentComplete); // Aggiorna la barra di progresso con la percentuale calcolata
    }
}
```

2.6.4. handleReadyStateChange(xhr, url)

Funzionamento: Gestisce i cambiamenti di stato della richiesta XMLHTTP. Se la richiesta è completata (`XMLHttpRequest.DONE`), controlla il codice di stato. Se è `200`, verifica che il tipo di contenuto sia `application/xml` e procede con l'elaborazione della risposta tramite processXMLResponse. In caso di risposta non valida o codice di stato diverso da `200`, segnala l'errore tramite `handleGlobalError`.

Codice Javascript

```
function handleReadyStateChange(xhr, url) {
    if (xhr.readyState === XMLHttpRequest.DONE) { // Controlla se la richiesta è stata completata
        if (xhr.status === 200) { // Controlla se la richiesta è stata completata con successo
            if (xhr.getResponseHeader("Content-Type").includes("application/xml")) {
                console.log(xhr.responseText); // Log del testo della risposta
                processXMLResponse(xhr.responseText); // Processa la risposta XML
            } else {
                handleGlobalError("Invalid response type: not XML", url); // Gestisci errori di tipo di risposta
            }
        } else {
            handleGlobalError("HTTP error: Status " + xhr.status, url); // Gestisci errori di stato HTTP
        }
    }
```

2.6.5. handleGlobalError(errorMsg, url, retryCallback)

Funzionamento: Registra l'errore nella console e lo mostra all'utente tramite `showErrorToUser`. Se è presente una funzione di ritentativo (`retryCallback`), la esegue. Questa funzione centralizza la gestione degli errori e l'interazione con l'utente.

Codice Javascript

```
function handleGlobalError(errorMsg, url = '', retryCallback = null) {
    console.error(`Errore: ${errorMsg}, URL: ${url}`); // Log dell'errore
    showErrorToUser(`Si è verificato un errore: ${errorMsg}`); // Mostra l'errore all'utente

    if (retryCallback) {
        // Opzionale: aggiungi logica per il tentativo di ritentare l'azione
        // Puoi anche passare il numero di tentativi rimasti se necessario
        retryCallback();
    }
}
```

2.6.6. updateProgressBar(percentage)

Funzionamento: Aggiorna visivamente la barra di progresso basandosi sulla percentuale fornita. Modifica gli attributi aria-valuenow, style, e il testo interno dell'elemento della barra di progresso per riflettere lo stato corrente del download.

Codice Javascript

```
function updateProgressBar(percentage) {
    const pb = document.getElementById("progressbar"); // Ottieni l'elemento della barra di progresso
    pb.setAttribute("aria-valuenow", percentage.toString()); // Imposta il valore attuale per l'accessibilità
    pb.setAttribute("style", "width: " + percentage.toString() + "%"); // Aggiorna la larghezza della barra
    pb.innerText = percentage.toFixed(1).toString() + "%"; // Mostra la percentuale come testo
}
```

2.6.7. retryRequest(url, attempts)

Funzionamento: Esegue tentativi ripetuti di eseguire readXmlFile in caso di fallimento della richiesta. Attende 2 secondi tra un tentativo e l'altro e decrementa il contatore dei tentativi rimasti. Se i tentativi si esauriscono, segnala l'errore tramite `handleGlobalError`.

Codice Javascript

```
function retryRequest(url, attempts) {
    if (attempts <= 0) {
        handleGlobalError("Tentativi esauriti per la richiesta", url); // Gestisci l'esaurimento dei tentativi
        return;
    }
    console.log(`Riprova richiesta per: ${url}, tentativi rimasti: ${attempts}`); // Log del tentativo di ripetizione
    setTimeout(function() {
        readXmlFile(url); // Ripeti la richiesta dopo un intervallo
    }, 2000);
    attempts--; // Decrementa il numero di tentativi rimasti
}
```

2.6.8. processXMLResponse(responseText)

Funzionamento: Processa la risposta XML. Rimuove gli spazi bianchi inutili, converte il testo in un documento XML, valida il documento tramite `validateXML`, e poi elabora i nodi XML per costruire e popolare una tabella HTML tramite createTable e populateTable.

Codice Javascript

```
function processXMLResponse(responseText) {
    try {
        const ris = responseText.replace(/>\s+<"); // Rimuovi gli spazi bianchi tra i tag
        const xml = new DOMParser(); // Crea un nuovo DOMParser
        const xmlDOC = xml.parseFromString(ris, "text/xml"); // Analizza la stringa di risposta come XML
        validateXML(xmlDOC); // Valida il documento XML

        const nodes = xmlDOC.documentElement.childNodes; // Ottieni i nodi figlio dell'elemento radice del documento
        const tbody = createTable(nodes); // Crea una tabella con i nodi
        populateTable(nodes, tbody); // Popola la tabella con i nodi
    } catch (error) {
        console.error("Errore nel processing della risposta XML: ", error); // Gestisci eventuali errori nel processing della risposta
    }
}
```

2.6.9. validateXML(xmlDOC)

Funzionamento: Valida il documento XML ricevuto. Verifica l'esistenza dell'elemento radice e l'assenza di errori. Controlla anche per specifici errori di parsing. Se il documento non è valido, lancia un errore.

Codice Javascript

```
function validateXML(xmlDOC) {
// Aggiungi qui la logica per verificare il contenuto del documento XML
// Ad esempio, controlla se i nodi specifici o gli attributi sono presenti
if (!xmlDOC.documentElement || xmlDOC.getElementsByTagName('errore').length > 0) {
    throw new Error("Documento XML non valido o contiene elementi di errore"); // Gestisci documenti XML non validi o con errori
}
if (xmlDOC.documentElement.nodeName === "parsererror") {
    throw new Error("Errore nel parsing XML: Contenuto non valido"); // Gestisci errori di parsing del documento XML
}
}
```

2.6.10. showErrorToUser(errorMessage)

Funzionamento: Mostra un messaggio di errore all'utente. Cerca un elemento nel DOM per visualizzare l'errore. Se l'elemento esiste, aggiorna il suo contenuto e lo rende visibile. In caso contrario, usa `alert` come fallback per mostrare l'errore.

Codice Javascript

```
function showErrorToUser(errorMessage) {
    const errorDiv = document.getElementById("error-message"); // Ottieni l'elemento per mostrare i messaggi di errore
    if (errorDiv) {
        errorDiv.innerText = errorMessage; // Imposta il testo del messaggio di errore
```

```
            errorDiv.style.display = "block"; // Rendi visibile il messaggio di errore
        } else {
            // Opzionale: considera un fallback, come un alert, se l'elemento del DOM non esiste
            alert(errorMessage); // Mostra un messaggio di errore con un alert
        }
    }
```

2.6.11. createDOMElement(tagName, attributes, textContent)

Funzionamento: Crea un elemento DOM del tipo specificato, assegna gli attributi forniti e imposta il contenuto testuale. Restituisce l'elemento creato, permettendo la sua ulteriore manipolazione o inserimento nel DOM.

Codice Javascript

```
function createDOMElement(tagName, attributes, textContent) {
    const element = document.createElement(tagName); // Crea un nuovo elemento con il nome del tag specificato
    if (attributes) {
        for (const [key, value] of Object.entries(attributes)) {
            element.setAttribute(key, value); // Imposta gli attributi sull'elemento
        }
    }
    if (textContent) {
        element.textContent = textContent; // Imposta il contenuto testuale dell'elemento
    }
    return element; // Ritorna l'elemento creato
}
```

2.6.12. createTable(nodes)

Funzionamento: Crea una tabella HTML basandosi sui nodi XML forniti. Costruisce l'intestazione della tabella e il corpo utilizzando `createDOMElement`. Usa `DocumentFragment` per m`inimizzare il reflow e il repaint` durante la costruzione della tabella.

Codice Javascript

```
function createTable(nodes) {
try {
    if (!nodes || nodes.length === 0) {
        throw new Error("Nessun nodo fornito per creare la tabella."); // Gestisci il caso in cui non ci sono nodi da elaborare
    }

    const div = document.getElementById("xmlTable"); // Ottieni l'elemento div dove inserire la tabella
    if (!div) {
        throw new Error("Elemento div per la tabella non trovato."); // Gestisci il caso in cui l'elemento div non è trovato
    }

    const table = document.createElement("table"); // Crea l'elemento della tabella
    table.setAttribute("class", "table table-striped table-hover"); // Imposta le classi CSS sulla tabella

    const thead = document.createElement("thead"); // Crea l'elemento dell'intestazione della tabella
    const tbody = document.createElement("tbody"); // Crea l'elemento del corpo della tabella
    const tr = document.createElement("tr"); // Crea l'elemento della riga per l'intestazione della tabella
    const fragment = document.createDocumentFragment();  // Crea un DocumentFragment per ottimizzare l'inserimento nel DOM

    table.appendChild(thead); // Aggiungi l'intestazione alla tabella
    table.appendChild(tbody); // Aggiungi il corpo alla tabella
    thead.appendChild(tr); // Aggiungi la riga all'intestazione

    // Controlla se il primo nodo ha figli e attributi prima di costruire le intestazioni
    if (nodes[0] && nodes[0].childNodes) {
        // Combina la creazione delle intestazioni per childNodes e attributes
        [...nodes[0].childNodes, ...nodes[0].attributes].forEach(item => {
            const th = createDOMElement("th", {"scope": "col"}, item.nodeName); // Crea un'intestazione per ogni nodo o attributo
            fragment.appendChild(th);  // Aggiungi l'intestazione al fragment
            tr.appendChild(th); // Aggiungi l'intestazione alla riga
        });
    } else {
        throw new Error("Il primo nodo non ha figli o attributi."); // Gestisci il caso in cui il primo nodo non ha figli o attributi
    }

    div.appendChild(table); // Aggiungi la tabella al div

    return tbody; // Ritorna l'elemento del corpo della tabella per ulteriori operazioni
} catch (error) {
    console.error("Errore nella creazione della tabella: ", error); // Gestisci eventuali errori nella creazione della tabella
    // Gestisci ulteriormente l'errore, come mostrare un messaggio all'utente
```

```
    }
}
```

2.6.13. populateTable(nodes, tbody)

Funzionamento: Popola il corpo della tabella (`tbody`) con i dati forniti (`nodes`). Crea righe e celle per ogni nodo e attributo XML. Anche qui, utilizza `DocumentFragment` per ottimizzare l'inserimento dei nodi nel DOM.

Codice Javascript

```
function populateTable(nodes, tbody) {
    try {
        if (!nodes || nodes.length === 0) {
            throw new Error("Nessun nodo fornito per popolare la tabella."); // Gestisci il caso in cui non ci sono nodi da inserire nella tabella
        }
        if (!tbody) {
            throw new Error("Elemento tbody non fornito."); // Gestisci il caso in cui l'elemento del corpo della tabella non è fornito
        }

        // Crea un DocumentFragment per contenere temporaneamente le righe della tabella
        const fragment = document.createDocumentFragment();

        for (let item of nodes) {
            const tr = document.createElement("tr"); // Crea una nuova riga per la tabella
            for (let subItem of [...item.childNodes, ...item.attributes]) {
                const td = document.createElement("td"); // Crea una nuova cella per la riga
                td.innerText = subItem.textContent || ''; // Imposta il testo della cella
                tr.appendChild(td); // Aggiungi la cella alla riga
            }
            fragment.appendChild(tr); // Aggiungi la riga al fragment anziché direttamente al tbody
        }
        tbody.appendChild(fragment); // Aggiungi tutti i nodi al tbody in un solo reflow/repaint
    } catch (error) {
        console.error("Errore nel popolare la tabella: ", error); // Gestisci eventuali errori nel popolamento della tabella
        // Gestisci ulteriormente l'errore, come mostrare un messaggio all'utente
    }
}
```

2.6.14. Conclusioni

Il codice mostra un'attenta considerazione per la robustezza, l'esperienza utente e l'ottimizzazione delle prestazioni, evidenziata dalla gestione dettagliata delle richieste, dalla validazione dei dati, dalla creazione dinamica del DOM e dalla centralizzazione della gestione degli errori.