Documentazione tecnica

Premessa questo documento contiene solo le spiegazioni riguardante il codice sorgente sviluppato per questa applicazione.

Shop.html

Response.sort((a, b) => a.credits - b.credits); Riga

Il metodo `.sort()` in JavaScript ordina in loco un array di oggetti, restituendo l'array ordinato. La funzione di confronto `(a, b) => a.credits - b.credits` ordina gli oggetti in base al valore dell'attributo `credits`. Se la differenza è negativa, `a` precede `b`; se positiva, `b` precede `a`. Un risultato di zero mantiene l'ordine originale tra `a` e `b`, garantendo un ordinamento stabile. Dopo l'esecuzione, l'array `response` sarà ordinato per valori crescenti di `credits`.

La funzione createCard() viene utilizzata per generare dinamicamente le card e gestirne la visualizzazione. Se si clonassero le card esistenti impostandole a display: none, rimarrebbero comunque nel DOM, risultando visibili (anche se vuote o bianche). La funzione permette di creare le card solo quando necessario, evitando elementi vuoti o invisibili nel DOM. Anche se implementata in modo diverso in trade.js, la logica di base serve lo stesso scopo: creare e gestire le card in modo efficiente senza problemi di visualizzazione incompleta o non desiderata.

```
 //spiegazione riguardante il codice
generateDiscount()
      .then(() => generateCredits())
      .then(() => generateAlbums())
      .then(() => showCart())
      .catch(error => {
         console.error("Error in fetching data:", error);
   });
```

Ho gestito le funzioni **generateAlbums**, **generateCredits** e **showCart** come Promises per controllare il flusso di esecuzione ed evitare richieste concorrenti in MongoDB.

Attraverso la funzione generateDiscount, restituisco il risultato della prima fetch come Promise. Questo assicura che solo dopo il completamento della prima richiesta, si avvii la seconda, prevenendo problemi di concorrenza.

Ho scelto di caricare prima lo **shop** e poi il **carrello**, in modo che, in caso di latenza del server, le informazioni più rilevanti vengano caricate per prime.

(deriva da un errore che era causato dal continuo aprire e chiudere il sever di mongo per fare richieste che poi è stato risolto in un altro modo, ma il codice di fatto non l'ho più toccato perchè countinua a funzionare)

Globals.js

la funzione getRandomInt() funziona nel seguente modo:

Math.random(): Questa funzione restituisce un numero pseudo-casuale in virgola mobile compreso tra 0 (incluso) e 1 (escluso).

Math.random() * (max - min + 1): Moltiplica il valore restituito da Math.random() per (max - min + 1) facendo uscire un numero con la virgola.

La differenza (max - min + 1) rappresenta l'ampiezza dell'intervallo tra min e max, inclusi i limiti che viene moltiplicata per il range a 0 a 1 del numero

Math.floor(): Arrotonda il numero risultante verso il basso al numero intero più vicino, riducendo il range a [0, max - min].

+ min: Trasla l'intervallo generato da [0, max - min] a [min, max], garantendo che il numero casuale risultante sia compreso tra min e max, inclusi.

const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

/: Inizio e fine della stringa

^: indicare che la stringa deve cominciare con il pattern che segue.

[^\s@]+: Cerca uno o più caratteri che non siano spazi o @.

- **^** dato che è usato all'interno di un array è il sibolo di negazione per cio che segue.
- **\s** rappresenta gli spazi.
- @ rappresenta la @

\.: Il punto (**.**) è un carattere speciale nelle regex, quindi deve essere preceduto da un backslash (\) per indicare che si sta cercando un punto letterale.

$/: Fine della stringa.

const passwordRegex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$/;

- **(?=.*[a-z])**: Asserzione di lookahead (lookahead assertion). Verifica che nella stringa ci sia almeno una lettera minuscola (a-z).
  - **(?=...)**: Lookahead, un controllo che richiede una certa condizione, senza consumare caratteri.
  - **[a-z]**: Almeno una lettera minuscola.
- **(?=.*[A-Z])**: Verifica che ci sia almeno una lettera maiuscola (A-Z).
- **(?=.*\d)**: Verifica che ci sia almeno una cifra (\d).
  - **\d**: Rappresenta una cifra da 0 a 9.
- **(?=.*[@$!%*?&])**: Verifica che ci sia almeno uno dei caratteri speciali tra @$!%*?&.
- **[A-Za-z\d@$!%*?&]{8,}**: La stringa deve essere composta solo da lettere maiuscole e minuscole (A-Za-z), cifre (\d), e simboli speciali (@$!%*?&). La lunghezza minima deve essere **8 caratteri**.
  - **{8,}**: Indica che la lunghezza deve essere di almeno 8 caratteri.

Trade.html

function findDuplicates(array) { //trova le figurine doppie che puoi scambiare

let duplicates = [];

for (let i = 0; i < array.length; i++) { //per ogni elemento dell'array

if (!duplicates.some(duplicate => duplicate.id === array[i])) { //se non è già stato inserito

let count = (array.filter(item => item === array[i]).length) - 1; //conto quante volte lo trovo nell'array e poi metto solo le ripetizioni

if (count > 0) {//se si ripete più di una volta lo inserisco

```
            duplicates.push({ id: array[i], count: count });
          }
        }
      }
      return duplicates;
  }
```

il metodo .some() controlla se esiste già un oggetto nell'array duplicates con un campo ID uguale all'elemento corrente array[i]. Se non esiste (!duplicates.some(...)) ovvero restituira false, il blocco di codice successivo viene eseguito.

Viene utilizzato il metodo .filter() per creare un nuovo array che contiene solo gli elementi uguali a array[i]. Poi, .length restituisce il numero di elementi in questo array, cioè quante volte l'elemento corrente appare nell'array originale. Questo numero viene salvato nella variabile count.

Node.js

```
process.on('SIGINT', async () => {
  try {
    await client.close();
    console.log("MongoDB connection closed");
    process.exit(0);
  } catch (err) {
    console.error("Failed to close MongoDB connection", err);
    process.exit(1);
  }
});
```

spiegazione:

- **process:** l'oggetto process è una variabile globale in Node.js che fornisce informazioni e controllo sul processo Node.js corrente. In Node.js, un "processo" si riferisce al processo di esecuzione di un'applicazione Node.js, quindi si riferisce al flusso del codice della mia applicazione
- **on**: Il metodo on è utilizzato per ascoltare eventi specifici sul processo. In questo caso, stiamo ascoltando l'evento 'SIGINT'.
- **'SIGINT'**: Questo è il segnale di interruzione, che viene comunemente inviato quando l'utente preme Ctrl+C nel terminale per interrompere un processo Node.js.

Questo codice è progettato per garantire che, quando l'applicazione Node.js viene interrotta (ad esempio, quando l'utente preme Ctrl+C), la connessione al database MongoDB venga chiusa in modo ordinato.

Utilizzando un gestore di eventi per il segnale 'SIGINT', il codice assicura che la connessione venga chiusa correttamente e che il processo Node.js termini in modo pulito, sia in caso di successo che di errore.

- **process.exit(0):** Questo comando termina il processo Node.js. Il codice di uscita 0 indica che il processo è terminato senza errori. Invia un segnale al sistema operativo che tutto è andato bene.
- **process.exit(1):** Questo comando termina il processo Node.js con un codice di uscita 1,che indica che il processo è terminato a causa di un errore.

In node js ho usato un main unico che serve per aprire e chiudere le comunicazioni con il server mongo una sola volta questo perchè migliora dimolto il tempo di risposta del server durante le fetch.

QUERY PARTICOLARI:

```
let words = await client.db("Marvel").collection("languages").aggregate([
        { $match: { language: language } },       // Filtra per linguaggio
        { $unwind: "$items" },                    // "Esplodi" l'array items per lavorare su un singolo oggetto dell'array
        { $match: { "items.page": page } },       // ottine un array filtrato dove sono presenti solo oggetti che hanno la pagina che si sta cercando
        { $project: { _id: 0, text: "$items.text" } } // Proietta solo il campo text
    ]).toArray();
```

La struttura a pipeline in MongoDB viene utilizzata per ordinare la sequenza delle operazioni di aggregazione, cioè organizza l'operazione aggragate in varie fasi stile catena di montaggio. è definita come un array di oggetti, dove ogni oggetto rappresenta una fase. Le fasi sono rappresentate da operatori di aggregazione, come $match, **$group, $sort, $project, $unwind**, ecc.

$match: (filtra un array) Filtra i documenti in base a una condizione specifica, simile a una query find(). È spesso usato per ridurre il set di dati iniziale.

$project:(usato per inviare solo alcuni campi invece che l'intero oggetto) Seleziona o modifica i campi dei documenti. Può essere usato per includere, escludere o ricalcolare campi.

$unwind è un'operazione nella pipeline che serve a "esplodere" o "decomporre" un array presente in un documento. Permette di lavorare con singoli elementi di un array come se fossero documenti separati corrisponde ad un ARRAY[i]

```
{
   "title": "Learn MongoDB",
   "authors": ["Alice", "Bob", "Charlie"] ->array da esplodere
}
```

quando scrivi il codice { $unwind: "$authors" }

Dopo l'operazione di $unwind, ottieni tre documenti separati, uno per ogni autore:

```
{
   "title": "Learn MongoDB",
   "authors": "Alice"
}

{
   "title": "Learn MongoDB",
   "authors": "Bob"
}

{
   "title": "Learn MongoDB",
```

```
    "authors": "Charlie"
}
```