EX - Task Manager Avanzato

Consegna

> **Repo:** ex-react-task-manager

Sei stato assunto per costruire un **Task Manager Avanzato**, un'app web che permette agli utenti di creare, modificare, organizzare ed eliminare task in modo intuitivo ed efficiente.
L'app dovrà supportare **filtri avanzati, ricerca ottimizzata, ordinamento e conferme di azione con modali**. Inoltre, dovrà garantire un'**esperienza fluida** con prestazioni ottimizzate.

📌 Milestone 1 - Setup e Routing

Clonare il backend del progetto, impostare il frontend con Vite e configurare il routing con **`react-router-dom`**.

1. **Clonare e avviare il backend:**
   - Per gestire i task, utilizzeremo un backend già pronto.
   - Cloniamo il repository

     `https://github.com/boolean-it/react-task-manager-back`

     e avviamo il server con:

     ```
     npm install
     npm run start
     ```

- Dopo qualche secondo, nel terminale apparirà un messaggio simile a:

  ✅ `Server in ascolto su http://localhost:3001`

  Questo URL dovrà essere utilizzato per configurare il frontend.

2. **Impostiamo il frontend:**
   - Creiamo il progetto con **Vite**.
   - Installiamo **`react-router-dom`** nel progetto.
   - Creiamo il router principale in `App.jsx` utilizzando `BrowserRouter`.
3. **Definiamo due pagine principali:**
   - **Lista dei Task** (`TaskList.jsx`) → mostrerà l'elenco dei task.
   - **Aggiungi Task** (`AddTask.jsx`) → conterrà il form per aggiungere un nuovo task.
4. **Aggiungere una barra di navigazione** con `NavLink`, per permettere all'utente di spostarsi tra le pagine.
5. **Definire le rotte con `Routes` e `Route`**, associando ogni percorso alla rispettiva pagina.

📌 Milestone 2 - Setup Context API e Fetch Iniziale

Creare un **contesto globale** per la gestione dei dati e recuperare la lista dei task dall'API.

1. **Salvare l'URL dell'API nel file `.env` del progetto frontend:**
   - Creare un file `.env` nella cartella del progetto frontend e aggiungere lo URL della API raccolto alla Milestone 1.
   - In questo modo, l'URL sarà accessibile in tutto il progetto senza doverlo scrivere manualmente nel codice.

2. **Creare un Context API (`GlobalContext`)** per gestire lo stato globale dell'applicazione.

3. **Definire uno `useState` all'interno del provider**, per memorizzare la lista dei task.

4. **Effettuare una richiesta `GET` a `/tasks`** al caricamento dell'app, utilizzando `useEffect`, e salvare i dati nello stato.

5. **Stampare in console i dati ricevuti** per verificare il corretto recupero delle informazioni.

6. **Rendere disponibile il `GlobalContext.Provider`** in `App.jsx`, avvolgendo l'intera applicazione.

📌 Milestone 3 - Lista dei Task (Pagina)

Visualizzare l'elenco dei task in una tabella e ottimizzare il rendering con `React.memo()`.

1. **Recuperare la lista dei task dal `GlobalContext`** e mostrarla nella pagina `TaskList.jsx`.
2. **Strutturare `TaskList.jsx` come una tabella**, con le intestazioni `Nome`, `Stato`, `Data di Creazione`.
3. **Creare un componente `TaskRow.jsx`**, che rappresenta una singola riga della tabella e mostra solo le proprietà `title`, `status` e `createdAt` (escludendo `description`).
4. **Applicare uno stile differente alla colonna `status`**, assegnando i seguenti colori di sfondo alle celle in base al valore dello stato:
   - `"To do"` → **rosso**
   - `"Doing"` → **giallo**
   - `"Done"` → **verde**
5. **Utilizzare `React.memo()` su `TaskRow.jsx`** per ottimizzare le prestazioni ed evitare render inutili.

📌 Milestone 4 - Creazione del Custom Hook useTasks() (GET)

Creare un **custom hook** per centralizzare la gestione dei task e semplificare l'accesso ai dati.

1. **Creare un hook `useTasks()`** che recupera i task iniziali con una richiesta GET a `/tasks` e li memorizza in uno stato locale (`useState`).
2. **Definire le funzioni `addTask`, `removeTask`, `updateTask`** all'interno di `useTasks()`, lasciandole vuote per ora.
3. **Rendere disponibili le funzioni e la lista dei task** restituendole come valore dell'hook.
4. **Integrare `useTasks()` nel `GlobalContext`**, in modo che tutti i componenti possano accedere ai task e alle funzioni di gestione.

📌 Milestone 5 - Creazione del Form per Aggiungere un Task

Creare un form per aggiungere un task, senza ancora inviare i dati all'API.

1. **Aggiornare la pagina `AddTask.jsx`** per contenere un form con i seguenti campi:
   - **Nome del task (`title`)** → Input **controllato** (`useState`).
   - **Descrizione (`description`)** → **Textarea non controllata** (`useRef`).
   - **Stato (`status`)** → **Select non controllata** (`useRef`), con opzioni `"To do"`, `"Doing"`, `"Done"`, e valore predefinito `"To do"`.

2. **Validare il campo Nome (`title`)**:
   - Il campo **non può essere vuoto**.
   - **Non può contenere simboli speciali**.
   - Se il valore è errato, mostrare un **messaggio di errore**.
   - Utilizzare una costante con i caratteri vietati:

   ```
   const symbols = "!@#$%^&*()-_=+[]{}|;:'\\",.<>?/`~";
   ```

3. **Gestione del Submit del Form**:
   - Al click del bottone "Aggiungi Task", il form deve **SOLO** stampare in console l'oggetto task con i valori inseriti (**NON** deve ancora essere inviata la richiesta all'API).

📌 Milestone 6 - Integrazione dell'API per Aggiungere un Task (POST)

Collegare il form di `AddTask` all'API e completare la funzione `addTask` in `useTasks()`.

1. **Completare la funzione `addTask` in `useTasks()`**:
   - La funzione deve ricevere un oggetto contenente le proprietà `title`, `description` e `status`.
   - Effettuare una chiamata API `POST /tasks`, inviando l'oggetto come `body` in formato JSON.
   - La chiamata API restituisce un oggetto con la seguente struttura:
     - In caso di successo:

     ```
     { success: true, task: /* la task creata */ }
     ```

     - In caso di errore:

     ```
     { success: false, message: "Messaggio di errore" }
     ```

- La funzione addTask deve controllare il valore di `success` nella risposta:
  - **Se `success` è `true`**, aggiornare lo stato globale aggiungendo la nuova task.
  - **Se `success` è `false`**, lanciare un errore con `message` come testo.

2. **Modificare la gestione del Submit del Form** in `AddTask.jsx`:
   - **Eseguire la funzione addTask di `useTasks()`**, passando l'oggetto con `title`, `description` e `status`.
   - **Se la funzione esegue correttamente l'operazione**:
     - Mostrare un **alert di conferma** dell'avvenuta creazione della task.
     - Resettare il form.
   - **Se la funzione lancia un errore:**
     - Mostrare un **alert con il messaggio di errore** ricevuto.

📌 Milestone 7 - Creazione della Pagina Dettaglio Task

Creare la pagina `TaskDetail.jsx`, che visualizza i dettagli di un task

1. **Aggiornare `TaskRow.jsx`**
   - Rendere il `title` un **link** a `/task/:id`, in modo che cliccando sul nome del task si venga reindirizzati alla pagina di dettaglio.

2. **Aggiornare `App.jsx` per aggiungere la rotta `TaskDetail.jsx`**
   - Aggiungere la rotta `/task/:id` che caricherà il componente `TaskDetail.jsx`.

3. **Creare `TaskDetail.jsx`** per mostrare:
   - **Nome (`title`)**
   - **Descrizione (`description`)**
   - **Stato (`status`)**
   - **Data di creazione (`createdAt`)**
   - Un bottone **"Elimina Task"**, che per ora stampa solo `"Elimino task"` in console.

📌 Milestone 8 - Funzione di Eliminazione Task (DELETE)

Aggiungere la funzionalità di eliminazione di un task con una chiamata API e aggiornare lo stato.

1. **Completare la funzione `removeTask` in `useTasks()`**:
   - La funzione deve ricevere un `taskId` e **effettuare una chiamata API `DELETE /tasks/:id`**.

- La chiamata API restituisce un oggetto con la seguente struttura:
  - In caso di successo:

    ```
    { success: true }
    ```

  - In caso di errore:

    ```
    { success: false, message: "Messaggio di errore" }
    ```

- La funzione `removeTask` deve controllare il valore di `success` nella risposta:
  - **Se `success` è `true`**, rimuovere il task dallo stato globale.
  - **Se `success` è `false`**, lanciare un errore con `message` come testo.

2. **Gestire l'eliminazione della task in `TaskDetail.jsx`**:
   - Al click su **"Elimina Task"**, chiamare `removeTask` passando l'`id` del task.
   - **Se la funzione esegue correttamente l'operazione**:
     - Mostrare un **alert di conferma** dell'avvenuta eliminazione.
     - **Reindirizzare l'utente alla lista dei task (`/`)**.
   - **Se la funzione lancia un errore:**
     - Mostrare un **alert con il messaggio di errore** ricevuto.

📌 Milestone 9 - Componente Modal e Conferma Eliminazione Task

Creare un componente **Modal** riutilizzabile e utilizzarlo per confermare l'eliminazione di un task.

1. **Creare il componente `Modal.jsx`**, che deve:
   - Accettare i seguenti **props**:
     - `title`: il titolo della modale.
     - `content`: il contenuto principale della modale.
     - `show`: stato booleano per mostrare o nascondere la modale.
     - `onClose`: funzione per chiudere la modale.
     - `onConfirm`: funzione eseguita al click del bottone di conferma.
     - `confirmText` (opzionale, default `"Conferma"`): testo del bottone di conferma.
   - **Utilizzare `ReactDOM.createPortal`** per rendere la modale indipendente dal flusso di rendering.
   - Implementare i pulsanti "Annulla" (chiude la modale) e "Conferma" (esegue `onConfirm`).

2. **Integrare il componente Modal in `TaskDetail.jsx`** per confermare l'eliminazione:
   - Quando l'utente clicca su **"Elimina Task"**, deve aprirsi la modale di conferma.
   - Se l'utente conferma, vengono eseguite le stesse operazioni della **Milestone 8**.

📌 Milestone 10 - Modale e Funzione di Modifica Task (PUT)

Creare una modale per modificare i dettagli di un task e aggiornare i dati tramite API.

1. **Completare la funzione `updateTask` in `useTasks()`**:
   - La funzione deve ricevere un oggetto `updatedTask` e **effettuare una chiamata API `PUT /tasks/:id`**.
   - La chiamata API restituisce un oggetto con la seguente struttura:
     - In caso di successo:

       ```
       { success: true, task: /* la task aggiornata */ }
       ```

     - In caso di errore:

       ```
       { success: false, message: "Messaggio di errore" }
       ```

   - La funzione `updateTask` deve controllare il valore di `success` nella risposta:
     - **Se `success` è `true`**, aggiornare la task nello stato globale.
     - **Se `success` è `false`**, lanciare un errore con `message` come testo.

2. **Creare il componente `EditTaskModal.jsx`**:
   - Deve accettare i seguenti **props**:
     - `show` (boolean): determina se la modale è visibile.
     - `onClose` (function): funzione per chiudere la modale.
     - `task` (object): oggetto che rappresenta il task da modificare.
     - `onSave` (function): funzione che viene chiamata al salvataggio con il task aggiornato.
   - **Utilizzare il componente `Modal`** per creare la modale di modifica, passandogli i seguenti valori:
     - `title`: **"Modifica Task"**.
     - `content`: un form contenente i campi del task da modificare.
     - `confirmText`: **"Salva"**.
     - `onConfirm`: deve attivare il **submit del form**.

> 💡 **Importante:**
> - Per attivare il **submit del form**, dobbiamo ottenere un riferimento diretto al form all'interno del componente. Creiamo una **ref con `useRef()`** e associamola al form.
> - Questo ci permette di chiamare il metodo **`editFormRef.current.requestSubmit()`** quando l'utente clicca su "Salva" nella modale, simulando il comportamento di un normale submit.

- **Strutturare il form all'interno della modale**, includendo i seguenti campi:
  - **Nome (`title`)** → Input di testo **controllato** (`useState`).
  - **Descrizione (`description`)** → Textarea **controllata** (`useState`).
  - **Stato (`status`)** → Select **controllata** (`useState`) con opzioni `"To do"`, `"Doing"`, `"Done"`.
- **L'`onSubmit` del form deve eseguire `onSave`, passandogli la task modificata.**

3. **Integrare `EditTaskModal` in `TaskDetail.jsx`**, con un nuovo bottone **"Modifica Task"**:
   - Quando l'utente clicca su "Modifica", si apre la modale con il form precompilato.
   - L'`onSave` di `EditTaskModal` deve eseguire la funzione `updateTask` di `useTasks()`, passando la task modificata.
   - **Se la funzione esegue correttamente l'operazione**:
     - Mostrare un **alert di conferma** dell'avvenuta modifica.
     - Chiudere la modale.
   - **Se la funzione lancia un errore:**
     - Mostrare un **alert con il messaggio di errore** ricevuto.

📌 Milestone 11 - Ordinamento delle Task

Implementare un sistema di ordinamento nella tabella delle task, permettendo all'utente di ordinare i task in base a diversi criteri.

1. **Aggiungere due state in `TaskList.jsx`**:
   - `sortBy`: rappresenta il criterio di ordinamento (`title`, `status`, `createdAt`).
   - `sortOrder`: rappresenta la direzione (1 per crescente, –1 per decrescente).
   - Il **default** di **`sortBy`** è `createdAt`, il default di **`sortOrder`**, è 1.

2. **Modificare la tabella** per rendere cliccabili le intestazioni (`th`), in modo che al click:
   - Se la colonna è già selezionata (`sortBy` uguale alla colonna cliccata), invertire `sortOrder`.
   - Se la colonna è diversa, impostare `sortBy` sulla nuova colonna e `sortOrder` su 1.

3. **Implementare la logica di ordinamento con `useMemo()`**, in modo che l'array ordinato venga ricalcolato solo quando cambiano `tasks`, `sortBy` o `sortOrder`:
   - **Ordinamento per `title`** → alfabetico (`localeCompare`).
   - **Ordinamento per `status`** → ordine predefinito: `"To do"` < `"Doing"` < `"Done"`.
   - **Ordinamento per `createdAt`** → confrontando il valore numerico della data (`.getTime()`).

- **Applicare `sortOrder`** per definire se l'ordine è crescente o decrescente.

📌 Milestone 12 – Ricerca dei Task con Debounce

Aggiungere un **campo di ricerca** che permette all'utente di filtrare i task in base al nome, ottimizzando le prestazioni con **debounce**.

1. **Creare un input di ricerca controllato**
   - Aggiungere un input di ricerca **controllato** in `TaskList.jsx` sopra la tabella, in modo che l'utente possa digitare per cercare un task.
   - Creare uno stato `searchQuery` (`useState`) per memorizzare il valore dell'input.

2. **Modificare l'`useMemo()` per filtrare e ordinare i task**
   - **Applicare il filtraggio** basato su `searchQuery`.
   - La ricerca deve essere **case insensitive**.
   - Ordinare i risultati in base ai criteri esistenti (es. nome, stato, data di creazione).

3. **Aggiungere il debounce per migliorare le prestazioni**
   - Creare una funzione `debounce` con `setTimeout()` per ritardare l'aggiornamento di `searchQuery`.
   - Usare `useCallback()` per memorizzare la funzione di debounce e prevenire inutili ricalcoli.

> 💡 **Importante:**
> - Il debounce non funziona bene sugli input **controllati**.
> - **Rimuovere `value` dall'input**, rendendolo **non controllato**, affinché il debounce possa funzionare correttamente.

🎯 BONUS 1 – Selezione ed Eliminazione Multipla di Task

Implementare un sistema di **multi-selezione e cancellazione di Task**.

1. **Modificare `TaskRow.jsx` per supportare la selezione**
   - Aggiungere le prop:
     - `checked`: indica se la task è selezionata.
     - `onToggle`: funzione chiamata quando la checkbox viene cliccata.
   - Inserire una **checkbox** accanto al nome della task, controllata tramite `checked` e che richiama `onToggle(task.id)`.

2. **Gestire lo stato `selectedTaskIds` in `TaskList.jsx`**
   - Creare uno stato con `useState([])` per memorizzare gli ID delle task selezionate.
   - Creare una funzione `toggleSelection(taskId)` che aggiorna `selectedTaskIds`, aggiungendo o rimuovendo l'ID della task.
   - Passare a `TaskRow` il `checked` e `toggleSelection` come `onToggle`.

3. **Mostrare il pulsante "Elimina Selezionate"**
   - Mostrare un pulsante **"Elimina Selezionate"** solo se **l'array `selectedTaskIds` contiene almeno un elemento**.

4. **Implementare la funzione `removeMultipleTasks` in `useTasks.js`**
   - La funzione deve ricevere un array di ID e inviare più richieste **`DELETE /tasks/{id}`** in parallelo.
   - Utilizzare **`Promise.allSettled()`** per gestire successi ed errori senza interrompere il processo.
   - Per ogni richiesta che ha successo, **la task deve essere rimossa dallo stato locale**.
   - Se almeno una richiesta fallisce, **lanciare un errore** con un messaggio che mostra gli ID non eliminati.

5. **Gestire l'eliminazione multipla in `TaskList.jsx`**
   - Al click su **"Elimina Selezionate"**, chiamare **`removeMultipleTasks`** passando **`selectedTaskIds`**.
   - **Se la funzione esegue correttamente l'operazione**:
     - Mostrare un **alert di conferma** dell'avvenuta eliminazione multipla.
     - Svuotare **`selectedTaskIds`.**
   - **Se la funzione lancia un errore:**
     - Mostrare un **alert con il messaggio di errore** ricevuto.

🎯 BONUS 2 - Funzionalità Aggiuntive

Aggiungere funzionalità di personalizzazione, formattazione, validazione e centralizzazione dello stato.

1. **Usare `dayjs` per formattare le date in formato italiano (`DD/MM/YYYY`)**
   - **Installare `dayjs`** con il comando:

     ```
     npm install dayjs
     ```

   - **Modificare `TaskRow.jsx` e `TaskDetail.jsx`** per visualizzare la data formattata in formato italiano (`DD/MM/YYYY`).

2. **Aggiornare `addTask` e `updateTask` in `useTasks.js`** in modo che:
   - Prima di effettuare la chiamata API, controllino se esiste già un task con lo stesso nome.
   - Se il nome è già presente, **lanciare un errore e impedire la creazione/modifica**.

3. **Implementare `useReducer` per gestire lo stato dei task**
   - **Sostituire `useState` con `useReducer`** in `useTasks.js`.
   - **Creare un `tasksReducer.js`** per gestire le azioni (`LOAD_TASKS`, `ADD_TASK`, `REMOVE_TASK`, `UPDATE_TASK`, `REMOVE_MULTIPLE_TASKS`).
   - **Modificare tutte le funzioni (`addTask`, `removeTask`, `updateTask`, `removeMultipleTasks`) e il fetch iniziale** per aggiornare lo stato attraverso il reducer.