INTRODUZIONE

Homeganize è un'applicazione web moderna progettata per semplificare la gestione delle attività domestiche all'interno di un gruppo, sia esso composto da membri di una famiglia o da coinquilini. Basata sullo stack MERN (MongoDB, Express, ReactJS, NodeJS), Homeganize sfrutta un'architettura MVC (Model View Controller) organizzata su tre livelli logici.

Il frontend, sviluppato con ReactJS, si occupa della presentazione e dell'interazione con l'utente, offrendo un'interfaccia fluida e reattiva tipica delle Single Page Applications (SPA). La logica applicativa risiede nel backend, costruito con NodeJS e Express, che gestisce il flusso dei dati e implementa funzionalità come la gestione dei task, notifiche in tempo reale e sicurezza. L 'archiviazione dei dati è affidata a MongoDB, un database NoSQL scalabile e flessibile, ospitato sui server Atlas.

In produzione, l'applicazione adotta una configurazione three-tier, separando logicamente e fisicamente i livelli di presentazione, logica e dati per garantire scalabilità e sicurezza. Durante lo sviluppo, i livelli frontend e backend condividono la stessa macchina, semplificando il testing e l'integrazione, mentre il database rimane distribuito e accessibile tramite Atlas.

GUIDA ALL'INSTALLAZIONE

Per avviare l'applicazione, è necessario configurare correttamente sia il backend che il frontend, assicurandosi di avere NodeJS installato sul sistema. Dopo aver scaricato il codice sorgente, aprire due terminali: uno posizionato nella directory del backend e l'altro in quella del frontend. In ciascun terminale, eseguire il comando npm install o, in alternativa, npm -i, per scaricare automaticamente tutte le dipendenze richieste.

Se l'installazione automatica dovesse fallire, è possibile procedere manualmente installando ciascuna dipendenza indicata qui sotto.

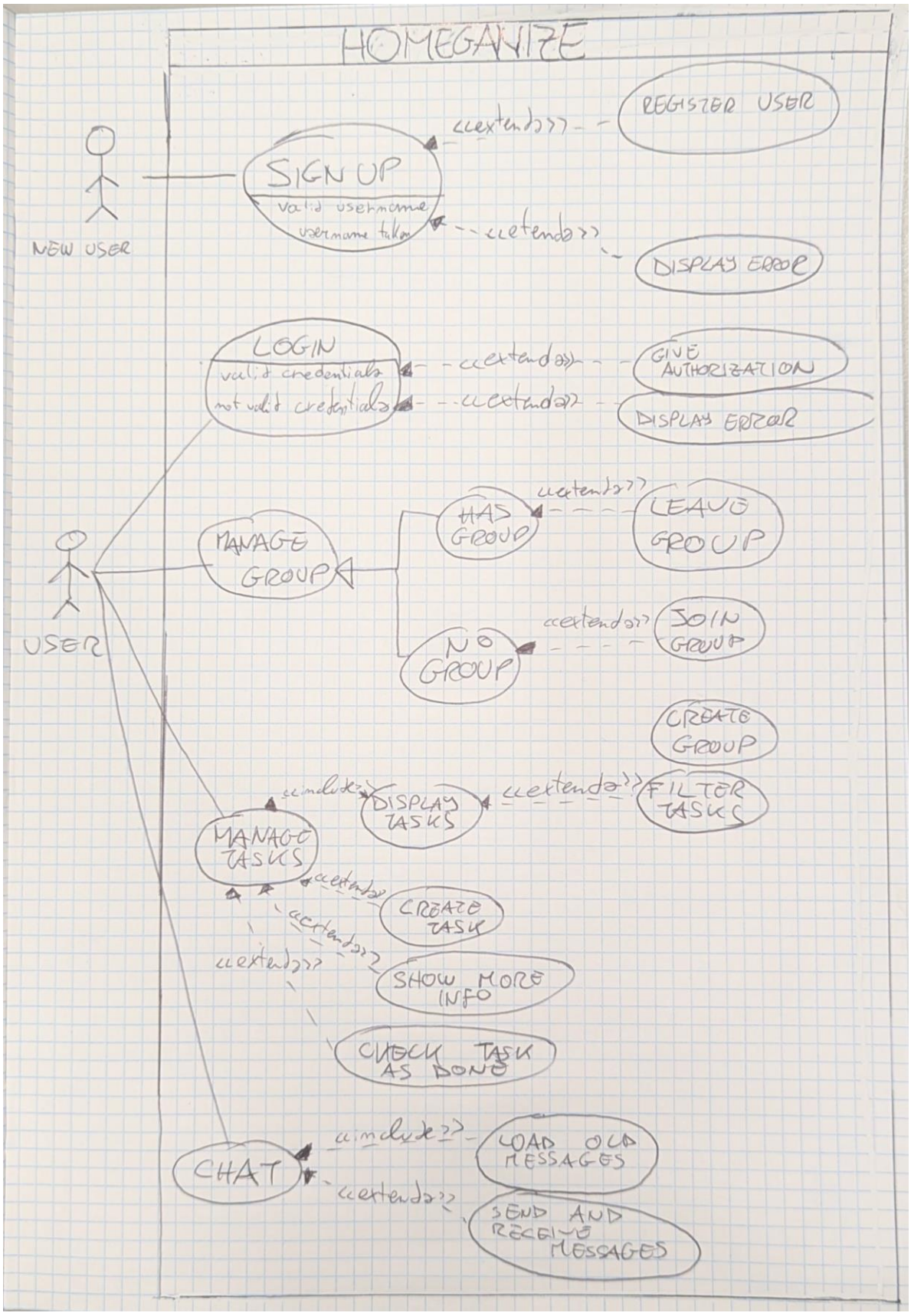
Frontend

- axios
- bootstrap
- bootstrap-icons
- date-fns
- express
- react
- react-bootstrap
- react-dom
- react-router
- react-router-dom
- react-scripts
- socket.io-client
- web-vitals

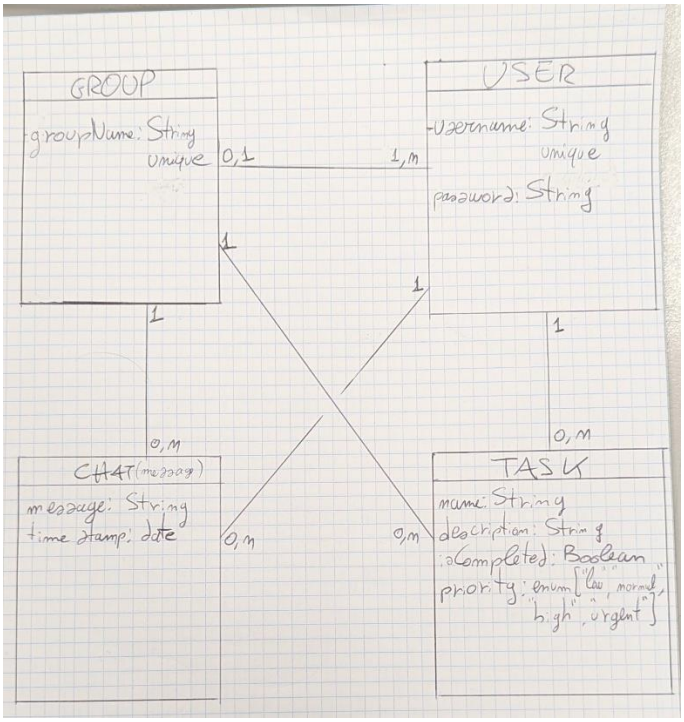
Backend

- axios
- bcrypt
- bcryptjs
- cookie-parser
- cors
- dotenv
- ejs
- express
- jsonwebtoken
- mongodb
- mongoose
- nodemon
- socket.io

CASI D'USO – UML DIAGRAM



MODELLI DEI DATI



User

Il modello UserModel rappresenta un utente dell'applicazione. Ogni utente ha un nome utente unico, una password, e può essere associato a un gruppo tramite una relazione di tipo ObjectId che punta al modello GroupModel. La password è configurata per essere esclusa dalla selezione predefinita, per ragioni di sicurezza, evitando che venga restituita insieme agli altri dati utente nelle query.

Task

Il modello TaskModel rappresenta un'attività che un utente può gestire. Ogni task ha un nome, una descrizione, un livello di priorità, e un flag che indica se è completato o meno. I task sono associati a un singolo utente e a un gruppo specifico. Le priorità del task sono limitate a un set definito di valori

Group

Il modello GroupModel rappresenta un gruppo di utenti. Ogni gruppo ha un nome e una lista di utenti e task associati. I gruppi permettono di organizzare le attività e di collaborare tra i membri del gruppo. Ogni gruppo può contenere più task e utenti, creando una relazione di tipo uno-a-molti con entrambi i modelli.

Chat

Il modello ChatModel è utilizzato per gestire i messaggi di chat all'interno di un gruppo. Ogni messaggio è associato a un gruppo specifico (groupId) e contiene informazioni sull'utente che ha inviato il messaggio, il contenuto del messaggio stesso e un timestamp..

FRONTEND

Index.js

Il file **index.js** è il punto di ingresso principale dell'applicazione React. In questo file vengono eseguite tutte le operazioni di inizializzazione necessarie per avviare l'applicazione. Importa le librerie di base di React e ReactDOM, quindi utilizza ReactDOM.render per montare l'app principale (solitamente definita come App) nell'elemento DOM con id "root". All'interno di questo file vengono inclusi eventuali provider di contesto o middleware necessari per il funzionamento dell'app, come ad esempio BrowserRouter per il routing. È anche possibile configurare eventuali librerie di styling o altri pacchetti di supporto globali, come Bootstrap, che vengono caricati a livello globale per garantire coerenza visiva nell'applicazione.

App.js

È il componente principale dell'applicazione frontend e gestisce il routing, l'autenticazione e la struttura complessiva dell'app. Implementa il meccanismo di routing con react-router-dom, che consente la navigazione tra diverse pagine (es. LoginPage, SignupPage, ecc.).

Variabili di stato:

- *loadingVerification*: booleano per gestire il caricamento dello stato di autenticazione.
- *isAuthenticated*: booleano che indica se l'utente è autenticato.
- *state*: oggetto contenente:
    - *loading*: indica se i dati del gruppo e degli utenti sono in fase di caricamento.
    - *users*: array di utenti caricati dal backend.
    - *group*: informazioni sul gruppo dell'utente autenticato.

Funzioni:

- *checkAuth()*: verifica lo stato di autenticazione tramite l'API verification().
- *fetchUsers()*: recupera gli utenti registrati nel sistema dal backend.
- *fetchGroup()*: recupera le informazioni sul gruppo dell'utente autenticato.

Valori di ritorno:
Renderizza l'interfaccia principale dell'applicazione:

- *Header* con pulsanti di login, logout o registrazione.
- *Sidebar* visibile solo agli utenti autenticati.

- Le diverse pagine, gestite da Routes, che includono:
    - Rotte pubbliche (es. login e registrazione).
    - Rotte protette (es. TasksPage, GroupPage, ChatPage).
    - Pagina di errore _404Page.

Header.js

Componente che rappresenta la barra di navigazione principale. Include il nome e il logo dell'app, il menu off-canvas per dispositivi mobili, e i pulsanti di login/logout.

Funzioni:

- *handleLogout()*: invoca l'API di logout, aggiorna lo stato di autenticazione e reindirizza l'utente alla pagina di login.
- Pulsanti per la navigazione tra le pagine di login, registrazione e logout.

Valori di ritorno:

- Mostra la barra di navigazione con il nome dell'app ("Homeganizer").
- Per gli utenti autenticati, include un menu laterale (off-canvas) per dispositivi mobili e un pulsante di logout.
- Per gli utenti non autenticati, mostra i pulsanti per login e registrazione.

Sidebar.js

Rappresenta il menu laterale disponibile per gli utenti autenticati. Consente la navigazione rapida tra le pagine principali dell'app.

Funzioni:

- Usa il metodo navigate di react-router-dom per reindirizzare l'utente alle seguenti pagine:
    - */tasks* per visualizzare e gestire le attività.
    - */group* per visualizzare o gestire il gruppo.
    - */chat* per accedere alla chat del gruppo.

Valori di ritorno:
Restituisce una lista verticale di pulsanti che reindirizzano alle rispettive pagine.

Pagine

Le pagine principali sono descritte in dettaglio di seguito:

- *LoginPage.jsx*:
  Permette agli utenti di autenticarsi utilizzando email e password. Effettua una chiamata API per autenticare l'utente e salvare il token.
- *SignupPage.jsx*:
  Consente agli utenti di registrarsi fornendo username, email e password. Al completamento, l'utente viene reindirizzato al login.
- *TasksPage.jsx*:
  Mostra le attività del gruppo. Consente di visualizzare, aggiungere o completare attività.
- *GroupPage.jsx*:
  Fornisce informazioni sul gruppo corrente dell'utente e consente di gestire membri o ruoli.
- *ChatPage.jsx*:
  Consente agli utenti di scambiare messaggi in tempo reale all'interno del gruppo.
- *_404Page.jsx*:
  Pagina di errore visualizzata quando un percorso non valido viene richiesto.

Login.jsx

Permette agli utenti di accedere all'applicazione tramite username e password.

Stati:

- *user*: oggetto che contiene i dati inseriti dall'utente (username e password).
- *error*: stringa per i messaggi di errore in caso di autenticazione fallita.
- *loading*: booleano per indicare se il processo di login è in corso.

Funzioni principali:

- *handleChange(e)*: aggiorna lo stato user in base ai valori inseriti nei campi del form.
- *handleSubmit(e)*: gestisce la sottomissione del form:
  - Effettua una chiamata API usando la funzione login.
  - Se l'accesso è riuscito (*status 200*):
    - Aggiorna lo stato *isAuthenticated*.
    - Reindirizza l'utente alla pagina */tasks*.
  - In caso di errore, mostra un messaggio nella variabile *error*.

Signup.jsx

Consente agli utenti di creare un account fornendo un username, una password e un indirizzo email.

Stati:

- *user*: oggetto che contiene i dati dell'utente (username, password, email).
- *error*: stringa per i messaggi di errore in caso di registrazione fallita.
- *loading*: booleano per indicare se il processo di registrazione è in corso.

Funzioni principali:

- *handleChange(e)*: aggiorna lo stato user in base ai valori inseriti nei campi del form.
- *handleSubmit(e)*: gestisce la sottomissione del form:
    - Effettua una chiamata API usando la funzione *signup*.
    - Se la registrazione ha successo (*status 201*):
    - Reindirizza alla pagina di login (*/*).
    - In caso di errore, aggiorna lo stato *error*.

auth.css

Le classi CSS sono definite nel file auth.css per standardizzare lo stile delle pagine di login e registrazione.

Struttura Generale

- .*auth-container*:
    - Imposta la larghezza massima del contenitore e lo centra verticalmente nella pagina.
    - Applica uno stile moderno con ombre e bordi arrotondati.
- .*auth-title*:
    - Stile del titolo della pagina, centrato e ben visibile.

Form

- *label*:
    - Stile semplice e leggibile per i campi del modulo.
- *input*:
    - Applica bordi arrotondati e un effetto di focus con il colore blu (#007bff).

- .auth-submit:

- Stile blu con effetto hover e stato disabilitato grigio.

Errori

- .error-message:
  - Mostra un box rosso per i messaggi di errore, con un testo chiaro e leggibile.

TaskPage.jsx

La componente TasksPage rappresenta una pagina principale per la gestione delle attività in un contesto collaborativo. Include funzionalità per visualizzare, filtrare, creare e aggiornare attività, con supporto per interazioni dinamiche basate sullo stato dell'applicazione.

Proprietà

La componente accetta le seguenti props:

- *group* (Object, obbligatorio): Il gruppo a cui appartengono le attività.
- *users* (Array, opzionale): Lista di utenti disponibile per assegnare le attività.

Stato Interno

- *tasksList* (Array): Lista delle attività recuperate dal server.
- *expandedTaskId* (String): ID dell'attività espansa per mostrare più dettagli.
- *isFilterTabOpen* (Boolean): Stato di visibilità del pannello di filtro.
- *selectedPriorityFilter* (String): Filtro attivo per priorità delle attività.
- *createTaskModalVisible* (Boolean): Stato di visibilità della finestra di creazione attività.
- *loading* (Boolean): Stato di caricamento per operazioni asincrone.

Effetti

1. Effetto per il recupero delle attività:
   - All'avvio della componente (*useEffect* senza dipendenze), vengono recuperate le attività con una richiesta asincrona utilizzando l'API getTasks.

Funzioni

- *handleTaskCreated*: Aggiorna la lista delle attività recuperando nuovamente i dati dal server dopo la creazione di una nuova attività.
- *handleMoreInfoClick(taskId)*: Espande o collassa i dettagli di un'attività specifica in base al suo ID.

- *handleVisibleCreateTask*: Alterna la visibilità del modal di creazione attività e aggiorna l'URL hash (#createtask).
- *handleTaskDone(taskId, isCompleted)*: Aggiorna lo stato di completamento di un'attività specifica.
- *handleFilterselectedClick(filter)*: Applica un filtro di priorità alle attività visualizzate.
- *renderTasks(isCompleted)*: Rende dinamicamente una lista di attività filtrata in base allo stato di completamento e alla priorità selezionata.

Metodo di Rendering

1. Se il caricamento è in corso (*loading* === *true*): Ritorna un messaggio di caricamento (*<h1>Loading...</h1>*).
2. Se il gruppo non è definito: Mostra un avviso indicando che l'utente deve appartenere a un gruppo.
3. Se i dati sono disponibili:
    - Include una barra di navigazione con due pulsanti:
        - Creazione attività: Apre il modal di creazione.
        - Filtraggio attività: Mostra/nasconde il pannello di filtro.
    - Mostra le attività organizzate in due sezioni:
        - Attività incomplete
        - Attività completate
    - Includere componenti secondarie come:
        - FilterTasks per il filtraggio.
        - CreateTask per la creazione.

TaskPage.jsx

La componente **TaskCard** rappresenta un'attività individuale nella lista. Ogni TaskCard include informazioni di base sull'attività, funzioni interattive per completare/annullare attività e un'area espandibile per visualizzare ulteriori dettagli.

Proprietà

La componente accetta le seguenti props:

- *task* (Object, obbligatorio):
    - Proprietà dell'oggetto task:
        - *_id* (String): Identificativo unico dell'attività.
        - *name* (String): Nome dell'attività.

        - *description* (String): Dettagli dell'attività.
        - *priority* (String): Livello di priorità dell'attività. Valori validi: *'urgent'*, *'high'*, *'normal'*, *'low'*.
        - *isCompleted* (Boolean): Stato di completamento dell'attività.
        - *user* (Object):
            - *username* (String): Nome dell'utente assegnato all'attività.
- *isExpanded* (Boolean): Determina se i dettagli aggiuntivi dell'attività sono visibili.
- *onMoreInfoClick* (Function, obbligatoria): Callback chiamata quando l'utente clicca per espandere/collassare l'attività.
- onTaskDone (Function, obbligatoria): Callback chiamata quando l'utente segna *un'attività* come completata o incompleta.

Stato Interno

- *stateIsCompleted* (Boolean): Stato interno che riflette il completamento dell'attività. Sincronizzato con la prop task.isCompleted.
- *loading* (Boolean): Stato di caricamento durante l'aggiornamento del completamento dell'attività.

Effetti

- Sincronizzazione dello stato *stateIsCompleted*:
    - Ogni volta che *task.isCompleted* cambia, l'effetto (*useEffect*) sincronizza lo stato interno stateIsCompleted con il valore aggiornato.

Funzioni

- *handleDoneClick*(*event*):
    - Alterna lo stato di completamento di un'attività (da completata a incompleta o viceversa).
    - Aggiorna il database tramite la funzione API *updateTask*.
    - Aggiorna il componente genitore tramite il callback *onTaskDone*.

Metodo di Rendering

- Stile dinamico della card:
    - Se l'attività è completata, viene aggiunta la classe CSS complete.
    - Se l'attività ha priorità *'urgent'*, viene evidenziata con la classe task-urgent.

    - Se l'attività è espansa, viene aggiunta la classe expanded; altrimenti *not-expanded*.

- Icone di interazione:
    - Completamento attività: Mostra un'icona di conferma (bi-check-circle-fill) o annullamento (bi-ban-fill).
    - Espansione dettagli: Mostra un'icona per espandere (bi-arrow-down) o collassare (bi-arrow-up) la descrizione.
- Contenuto espanso:
    - Se isExpanded è *true*, visualizza la descrizione dell'attività.

Comportamento CSS

- Classi CSS rilevanti:
    - *.card*: Stile generale della card.
    - *.complete*: Evidenzia le attività completate con uno sfondo verde.
    - *.task-urgent*: Evidenzia le attività urgenti con uno sfondo rosso.
    - *.not-expanded*: Limita l'altezza della card.
    - *.moreInfoConten*t: Mostra contenuto espanso con gestione del testo a capo e altezza massima.

CreateTask.jsx

La componente CreateTask consente di creare nuove attività. Fornisce un'interfaccia utente per inserire dettagli dell'attività, selezionare una priorità e assegnare un utente.

Proprietà

- *onVisibleCreateTask* (Function, obbligatoria): Callback per chiudere la finestra di creazione del task.
- *onTaskCreated* (Function, obbligatoria): Callback chiamata quando un'attività viene creata con successo.
- *users* (Array, obbligatoria): Lista degli utenti disponibili per assegnare l'attività. Ogni elemento è un oggetto con almeno:
    - _id (String): Identificativo unico dell'utente.
    - username (String): Nome dell'utente.

Stato Interno

- *task* (Object): Rappresenta i dettagli dell'attività in fase di creazione.
    - *name* (String): Nome dell'attività.
    - *assinedUsername* (String): Username della persona assegnata.

    - *description* (String): Descrizione dell'attività.
    - *priority* (String): Livello di priorità selezionato.
    - *isCompleted* (Boolean): Stato iniziale di completamento (di default false).
- *loading* (Boolean): Indica se l'attività è in fase di creazione.
- *error* (String o null): Messaggio di errore se la creazione fallisce.
- *success* (Boolean): Indica se l'attività è stata creata con successo.

Funzioni

- *handleChange*(event):
    - Aggiorna i valori del form in base ai dati inseriti dall'utente.
- *handleSubmit*(event):
    - Valida i dati del form e invia una richiesta per creare un nuovo task utilizzando l'API addTask.
    - Chiama i callback onTaskCreated e onVisibleCreateTask al completamento.

Metodo di Rendering

- Mostra un modulo con i seguenti campi:
    - Nome attività (Text input): Campo obbligatorio.
    - Utente assegnato (Select dropdown):
        - Popolato dinamicamente con i dati di users.
        - Disabilitato se la lista è vuota.
    - Descrizione (Textarea): Campo opzionale per i dettagli dell'attività.
    - Priorità (Radio buttons): Permette di selezionare tra 'urgent', 'high', 'normal', 'low'.
- Stato interattivo:
    - Disabilita il pulsante di invio mentre loading è attivo.
    - Mostra un messaggio di errore (error) o successo (success) in base all'esito.

Comportamento CSS

- *overlay-create-task*: Sfondo scuro semi-trasparente che copre la pagina.
- *create-task-window*: Finestra modale centrale con stile arrotondato.
- *close-btn*: Icona cliccabile per chiudere la finestra.

- *submit-btn*: Bottone di invio stilizzato.

FilterTask.jsx

La componente FilterTask consente di filtrare le attività in base al livello di priorità selezionato.

Proprietà

- *onFilterSelectedClick (Function, obbligatoria)*: Callback chiamata ogni volta che un filtro di priorità viene selezionato. Riceve il valore della priorità come argomento.

Stato Interno

- *priorityFilter (String)*: Stato locale che tiene traccia del filtro selezionato. I valori possibili sono:
    - *'urgent'*
    - *'high'*
    - *'normal'*
    - *'low'*
    - *'none'*

Funzioni

- *handleFilterChange*(newFilter):
    - Aggiorna lo stato interno priorityFilter con il filtro selezionato.
    - Invia il filtro selezionato al componente genitore tramite il callback onFilterSelectedClick.

Metodo di Rendering

- Mostra una serie di pulsanti per ciascun livello di priorità:
    - *URGENT, HIGH, NORMAL, LOW, none* (nessun filtro).
- Stile dinamico:
    - Il pulsante attivo viene evidenziato con la classe btn-dark.
    - I pulsanti inattivi utilizzano la classe btn-secondary.

Comportamento CSS

Stile dinamico:

- Se il filtro è selezionato: btn-dark.
- Altrimenti: btn-secondary.

GroupPage.jsx

La componente GroupPage gestisce la creazione, l'unione e l'abbandono di gruppi da parte dell'utente. L'interfaccia è dinamica e si adatta a seconda che l'utente appartenga o meno a un gruppo.

Proprietà

- *state* (Object, obbligatoria):
    - Contiene lo stato globale dell'app, incluso il gruppo a cui l'utente appartiene.
- *setState* (Function, obbligatoria):
    - Funzione per aggiornare lo stato globale dell'app.

Stato Interno

- *groupName* (String): Nome del gruppo o codice gruppo in input.
- *error* (String): Messaggio di errore in caso di operazione fallita.
- *success* (String): Messaggio di successo in caso di operazione completata.
- *loading* (Boolean): Indica se un'operazione è in corso.

Funzioni

handleChangeCreate(e)

- Aggiorna il valore di groupName in base all'input utente.
- Parametri:
    - e: Evento dell'input.

handleSubmitCreate(e)

- Gestisce la creazione di un nuovo gruppo.
- Chiama l'API createGroup e aggiorna lo stato globale con il nuovo gruppo.
- Parametri:
    - e: Evento del form submit.

handleSubmitJoin(e)

- Gestisce l'unione a un gruppo esistente tramite codice gruppo.
- Chiama l'API addGroupToUser e aggiorna lo stato globale con il nuovo gruppo.
- Parametri:
    - e: Evento del form submit.

handleLeaveGroup()

- Gestisce l'abbandono del gruppo corrente.
- Chiama l'API leaveGroup e aggiorna lo stato globale rimuovendo il gruppo.

Metodo di Rendering

Caso: Nessun gruppo attivo (state.group === null)

- Sezioni disponibili:
    1. Crea un Nuovo Gruppo:
        - Input per il nome del gruppo.
        - Bottone per inviare il modulo.
    2. Unisciti a un Gruppo Esistente:
        - Input per il codice del gruppo.
        - Bottone per inviare il modulo.
- Messaggi:
    - Mostra error o success al completamento dell'operazione.

Caso: Utente già in un gruppo (state.group !== null)

- Mostra:
    - Messaggio che indica il gruppo attuale.
    - Bottone per abbandonare il gruppo.
- Messaggi:
    - Mostra error o success al completamento dell'operazione.

Comportamento CSS

- *.group-container*
    - Stile centrato con bordi arrotondati e ombreggiatura leggera.
- *.group-form*
    - Input e bottoni stilizzati con transizioni fluide.
- *.group-button, .leave-button*
    - Bottoni colorati con effetto hover e stato disabilitato.
    - Colori distintivi: blu per la creazione/unione, rosso per l'abbandono.
- Messaggi:
    - Errore: .error-message (sfondo rosso chiaro).

- Successo: .success-message (sfondo verde chiaro).

ChatPage.jsx

La componente ChatPage gestisce la visualizzazione e l'invio dei messaggi all'interno di una chat di gruppo. La pagina consente anche di connettersi a un server Socket.io per ricevere i messaggi in tempo reale.

Proprietà

- *group* (Object, obbligatorio):
  - Contiene i dettagli del gruppo di chat.
- *users* (Array, facoltativo):
  - Lista degli utenti appartenenti al gruppo. Non è usata direttamente in questo componente ma può essere utile per altre logiche.

Stato Interno

- *message* (String): Il messaggio attualmente scritto dall'utente nel campo di input.
- *messages* (Array): Lista dei messaggi ricevuti e salvati nella chat.
- *username* (String | null): Il nome utente dell'utente corrente.
- *socket* (Object | null): Oggetto socket per la connessione a Socket.io.
- *loading* (Boolean): Stato che indica se i dati sono in fase di caricamento.

***Effetti* e Funzioni**

useEffect - Caricamento iniziale dei messaggi e connessione al socket

- Quando il componente viene montato, viene effettuata una chiamata per recuperare i messaggi esistenti e le informazioni sull'utente (getMessages e getMe).
- Si stabilisce una connessione Socket.io con il server, e l'utente viene automaticamente aggiunto al gruppo di chat attraverso l'emissione dell'evento join_group.

sendMessage()

- Funzione che invia un messaggio al server. Se il messaggio non è vuoto, invia il messaggio salvato nel database tramite l'API saveMessage e lo emette attraverso Socket.io.
- L'input dell'utente viene poi resettato dopo l'invio del messaggio.

useEffect - Scroll automatico alla fine della chat

- Ogni volta che i messaggi cambiano, il componente scrolla automaticamente fino all'ultimo messaggio utilizzando messagesEndRef.

Funzionalità di Socket.io

- Connessione: La connessione al server Socket.io viene stabilita all'interno di useEffect. Quando la connessione è avvenuta, l'utente viene aggiunto al gruppo tramite l'evento join_group.
- Ricezione messaggi: Quando il server invia un messaggio (receive_message), il messaggio viene aggiunto alla lista dei messaggi.
- Invio messaggi: Quando l'utente invia un messaggio, il messaggio viene salvato e poi emesso al server tramite l'evento send_message.

Metodo di Rendering

1. Se il gruppo non è definito:
    - Viene mostrato un messaggio che indica che l'utente deve far parte di un gruppo.
2. Durante il caricamento:
    - Viene mostrato il messaggio "Caricamento..." mentre i dati sono in fase di recupero.
3. Chat Display:
    - Mostra il nome del gruppo e la lista dei messaggi ricevuti.
    - I messaggi sono divisi tra quelli inviati dall'utente (my-message) e quelli inviati dagli altri utenti (other-message), con uno stile di colore diverso per ciascuno.
    - Ogni messaggio mostra anche un timestamp formattato con date-fns.
4. Campo di Input e Bottone di Invio:
    - Un campo di input permette all'utente di scrivere il messaggio.
    - Il bottone "Invia" o l'azione di premere "Enter" inviano il messaggio.

Comportamento CSS

- *.chat-container*
    - Stile centrato con bordi arrotondati, ombreggiatura leggera, e padding per un aspetto pulito.
- *.chat-messages*
    - Max altezza per un'area scrollabile, con i messaggi che si estendono verticalmente.
- *.message, .my-message, .other-message*
    - I messaggi inviati dall'utente sono colorati di verde chiaro, quelli inviati da altri utenti di grigio chiaro.

    - Ogni messaggio è separato visivamente con margini e padding.

- *.timestamp*
    - Mostra la data e l'orario del messaggio, con una dimensione del font più piccola e un colore grigio chiaro.
- *.chat-input*
    - Layout a flessibile per mantenere l'input e il bottone di invio affiancati.
    - L'input ha bordi arrotondati e il bottone ha un colore blu con effetto hover.

BACKEND

Server.js

Il file server.js è il punto di ingresso principale per l'applicazione backend. Gestisce le connessioni al database, la configurazione del server HTTP, la gestione delle route e le comunicazioni in tempo reale tramite **Socket.io**.

Configurazione del Middleware

- *CORS* (Cross-Origin Resource Sharing):
    - Il CORS è configurato per consentire richieste provenienti dal dominio http://localhost:3000 (dove è ospitato il frontend) e abilitare l'invio di credenziali come i cookie.
- *Cookie-Parser*:
    - Viene utilizzato per parsare i cookie nelle richieste HTTP. Questo è utile per gestire le sessioni utente, come ad esempio per la gestione dell'autenticazione.
- *Express JSON*:
    - Il middleware express.json() permette di analizzare i corpi delle richieste in formato JSON, che è il formato usato dalle richieste HTTP del frontend.
- *Router*:
    - Il file router.js è importato e le sue rotte vengono usate come il principale sistema di gestione delle richieste HTTP. Questo separa la logica del backend in moduli più facilmente gestibili.

Connessione al Database (MongoDB)

- Il backend si connette a un database MongoDB utilizzando Mongoose. La connessione è configurata tramite una variabile d'ambiente **MONGODB_URI**, che contiene l'URI del database MongoDB.
- La funzione connectionDB() stabilisce la connessione e gestisce eventuali errori di connessione.

Configurazione di Socket.io

- Il server **Socket.io** viene configurato per gestire le connessioni WebSocket. Le comunicazioni avvengono tra il server e i client in tempo reale.
- **Configurazione CORS per Socket.io**: La configurazione CORS è la stessa di quella del server HTTP, consentendo la connessione solo dal frontend http://localhost:3000.

Gestione degli Eventi Socket.io

Il server gestisce vari eventi provenienti dai client connessi tramite WebSocket. Di seguito sono descritti i principali eventi gestiti:

1. *connection*:
    - Quando un client si connette al server WebSocket, il server stampa un messaggio di log con l'ID della connessione.
    - Ogni connessione riceve un socket che può essere utilizzato per inviare e ricevere eventi specifici.
2. *join_group:*
    - Quando un client invia una richiesta per entrare in un gruppo, viene emesso l'evento join_group, con il relativo groupId.
    - Se il groupId è valido, il client viene "unito" al gruppo tramite il comando socket.join(groupId).
    - Se il groupId non è valido o non è stato passato, il server invia un errore al client.
3. *send_message:*
    - Quando un client invia un messaggio a un gruppo, viene emesso l'evento send_message con i dati del messaggio, che includono groupId, message, e username.
    - Prima di inviare il messaggio, il server verifica che il client sia effettivamente nella stanza del gruppo (con il comando socket.rooms.has(groupId)).
    - Se il client non è nella stanza, viene inviato un errore al client.
    - Se il client è nella stanza, il messaggio viene inviato a tutti i membri del gruppo tramite l'evento receive_message.
4. *disconnect:*
    - Quando un client si disconnette, viene emesso l'evento disconnect e viene stampato un log con l'ID della connessione.

Server HTTP

- Il server HTTP viene creato utilizzando il modulo http di Node.js e l'istanza di Express.js. Il server ascolta sulla porta 5000.
- Le rotte del server sono gestite tramite il router importato dal file router.js.

Router.js

Il file router.js gestisce tutte le rotte API del backend. Si occupa di instradare le richieste HTTP tra i vari controller (ad esempio, gestione utenti, gruppi, compiti, e chat) e di

applicare il middleware di autenticazione. Ogni rotta è configurata per una specifica funzionalità e può essere protetta tramite il middleware authenticateToken, che verifica la validità del token JWT (JSON Web Token) per autenticare le richieste.

Autenticazione tramite JWT

Tutte le rotte che richiedono un'azione protetta, come la gestione degli utenti, dei task, dei gruppi e della chat, sono protette dal middleware authenticateToken. Questo middleware esamina il token JWT fornito nelle intestazioni della richiesta per verificare che l'utente sia autenticato.

Se il token è valido, l'utente può accedere alla risorsa richiesta. In caso contrario, viene restituito un errore di autenticazione.

userController.js

Il file userController.js gestisce la logica per le operazioni relative agli utenti, inclusi la registrazione, il login, il logout, la gestione dei gruppi e il recupero dei dati utente. Le operazioni principali coinvolgono la creazione, l'autenticazione, l'autorizzazione tramite JWT, e l'aggiunta o la rimozione da gruppi. Vengono utilizzati bcrypt per la gestione delle password e jsonwebtoken per la gestione dei token JWT.

Funzioni del Controller

1. *signupUser*

- **Descrizione**: Crea un nuovo utente, validando i dati di input (username e password). La password viene criptata tramite bcrypt prima di essere salvata nel database.
- **Metodo HTTP**: POST
- **Endpoint**: /signup
- **Input richiesto**:
    - username (stringa): Nome utente dell'utente che si sta registrando.
    - password (stringa): Password dell'utente da criptare.
- **Risposta**:
    - **Successo**: { message: 'user signed up successfully' }
    - **Errore**: { message: 'username and password are required' } se i campi sono mancanti o { message: 'username already used' } se l'username è già in uso.

2. *loginUser*

- **Descrizione**: Autentica un utente confrontando la password inserita con quella salvata nel database (utilizzando bcrypt per il confronto). Se la password è corretta, genera un token JWT per l'autorizzazione dell'utente.

- **Metodo HTTP**: POST
- **Endpoint**: /login
- **Input richiesto**:
    - username (stringa): Nome utente dell'utente che sta cercando di effettuare il login.
    - password (stringa): Password dell'utente.
- **Risposta**:
    - **Successo**: { message: 'login successful for', username: 'username' } con un token JWT memorizzato nei cookie.
    - **Errore**: { message: 'username and password are required' } se i campi non sono forniti o { message: 'username and password not valid' } se le credenziali non sono corrette.

3. *logoutUser*

- **Descrizione**: Disconnette un utente rimuovendo il token JWT dal cookie, annullando così l'autorizzazione.
- **Metodo HTTP**: POST
- **Endpoint**: /logout
- **Risposta**:
    - **Successo**: { message: "Logout successful" }
    - **Errore**: { message: 'error during logout' } in caso di errore durante il logout.

4. *getGroup*

- **Descrizione**: Restituisce informazioni sul gruppo a cui l'utente appartiene, se presente. L'utente è autenticato tramite il token JWT.
- **Metodo HTTP**: GET
- **Endpoint**: /group
- **Risposta**:
    - **Successo**: { message: 'group found successfully', groupInfo, hasGroup: true } con informazioni sul gruppo.
    - **Errore**: { message: 'user or group not found' } o { message: 'group not found' } se il gruppo o l'utente non esistono.

5. *addGroupToUser*

- **Descrizione**: Aggiunge un utente a un gruppo specificato tramite il nome del gruppo. Il gruppo deve esistere e l'utente non deve essere già membro del gruppo.

- **Metodo HTTP**: PATCH
- **Endpoint**: /user/addGroupToUser
- **Input richiesto**:
    - groupName (stringa): Nome del gruppo a cui l'utente deve essere aggiunto.
- **Risposta**:
    - **Successo**: { message: "user added to the group", newGroup } con le informazioni aggiornate del gruppo.
    - **Errore**: { message: 'group name not found' } o { message: 'user already in the group' } se il gruppo non esiste o l'utente è già membro del gruppo.

6. *leaveGroup*

- **Descrizione**: Permette a un utente di lasciare il proprio gruppo. L'utente viene rimosso dal gruppo e dal proprio campo "group" nel database.
- **Metodo HTTP**: PATCH
- **Endpoint**: /group/leave
- **Risposta**:
    - **Successo**: { message: 'Left group successfully' }
    - **Errore**: { message: 'group not found' } o { message: 'user not found or generic user update error' } se l'utente o il gruppo non vengono trovati.

7. *getUsernameById*

- **Descrizione**: Recupera il nome utente basato su un userId. È utilizzato principalmente per ottenere il nome utente di un utente dato il suo identificatore.
- **Metodo HTTP**: GET
- **Endpoint**: /usernameById
- **Input richiesto**:
    - userId (stringa): ID dell'utente di cui ottenere il nome.
- **Risposta**:
    - **Successo**: { message: 'user info sent correctly', username: 'username' }
    - **Errore**: { message: 'server error' } se si verifica un errore durante il recupero del nome utente.

Flusso di Autenticazione con JWT

1. L'utente si registra tramite la rotta POST /signup.

2. Una volta registrato, l'utente può effettuare il login tramite POST /login, che restituirà un token JWT.
3. Il token JWT viene memorizzato nel cookie del client per l'autenticazione nelle richieste successive.
4. Le rotte protette richiedono che il client invii il token JWT nel cookie o nell'intestazione per l'autenticazione.

taskController.js

Il controller taskController.js gestisce le operazioni relative alla creazione, recupero e aggiornamento dei task (compiti). Le operazioni principali comprendono l'aggiunta di nuovi task, la visualizzazione di tutti i task di un gruppo specifico e l'aggiornamento delle informazioni sui task esistenti.

Funzioni del Controller

1. addTask

- **Descrizione**: Crea un nuovo task associato a un utente e al suo gruppo. Verifica se il task contiene dati validi e se l'utente ha un gruppo. Il task viene poi associato al gruppo dell'utente.
- **Metodo HTTP**: POST
- **Endpoint**: /tasks
- **Input richiesto**:
    - task (oggetto): Contiene i dettagli del task da creare, tra cui assignedUsername (il nome utente a cui il task è assegnato) e altre informazioni specifiche del task.
- **Risposta**:
    - **Successo**: { message: 'task created successfully', task: createdTask } con i dettagli del task creato.
    - **Errore**: { message: 'task data is required' } se i dati del task sono mancanti o { message: 'user or group not found' } se l'utente o il gruppo non sono trovati.

2. getTasks

- **Descrizione**: Recupera tutti i task assegnati a un gruppo di cui l'utente è membro. L'utente è autenticato tramite il token JWT. Se l'utente non ha un gruppo, la richiesta restituirà un errore.
- **Metodo HTTP**: GET
- **Endpoint**: /tasks

- **Risposta**:
    - **Successo**: { message: 'tasks found', tasksFound } con la lista dei task trovati.
    - **Errore**: { message: 'user or group not found' } se l'utente non ha un gruppo associato o { message: 'error' } se si verifica un errore nel recupero dei task.

3. updateTask

- **Descrizione**: Aggiorna un task esistente identificato dal suo id. Viene eseguito tramite una richiesta PATCH, e l'aggiornamento avviene con i nuovi dati inviati nel corpo della richiesta.
- **Metodo HTTP**: PATCH
- **Endpoint**: /tasks/:id
- **Input richiesto**:
    - id (stringa): L'ID del task da aggiornare.
    - update (oggetto): I dati di aggiornamento per il task (ad esempio, modifiche al titolo, descrizione, stato, ecc.).
- **Risposta**:
    - **Successo**: { updatedTask } con i dettagli aggiornati del task.
    - **Errore**: { message: 'Task not found' } se il task con l'ID specificato non è stato trovato o { message: 'Error updating task' } se si verifica un errore durante l'aggiornamento.

Flusso di Lavoro

1. **Aggiungere un Task**:
    - Il client invia una richiesta POST /tasks con i dati del task, incluso l'utente assegnato al task.
    - Se l'utente è associato a un gruppo, il task viene creato e associato al gruppo di quell'utente.
2. **Recuperare i Task**:
    - Il client invia una richiesta GET /tasks, e il server risponde con tutti i task associati al gruppo dell'utente autenticato.
3. **Aggiornare un Task**:
    - Il client invia una richiesta PATCH /tasks/:id con l'ID del task e i dati da aggiornare. Se il task esiste, i dettagli vengono aggiornati nel database.

groupController.js

Il controller groupController.js gestisce le operazioni relative alla creazione, all'aggiunta di utenti e alla visualizzazione degli utenti di un gruppo. Le operazioni principali includono la creazione di un gruppo, l'aggiunta di un utente a un gruppo e il recupero degli utenti di un gruppo.

Funzioni del Controller

1. *createGroup*

- **Descrizione**: Crea un nuovo gruppo e associa l'utente che sta creando il gruppo come membro iniziale. Se l'utente è già membro di un altro gruppo, l'operazione fallisce.
- **Metodo HTTP**: POST
- **Endpoint**: /group
- **Input richiesto**:
    - groupName (stringa): Il nome del nuovo gruppo.
- **Risposta**:
    - **Successo**: { message: 'group created successfully', newGroup } con i dettagli del nuovo gruppo creato.
    - **Errore**:
        - { message: "Group name is required" } se il nome del gruppo non è fornito.
        - { message: "UserId error" } se non viene fornito un userId valido.
        - { message: "cannot be in two groups simultaneously" } se l'utente è già membro di un altro gruppo.
        - { message: 'group already exists' } se il gruppo con lo stesso nome esiste già.

2. *joinGroup*

- **Descrizione**: Aggiunge un utente a un gruppo esistente. Controlla se il gruppo esiste, se l'utente non è già membro e se l'utente non è già in un altro gruppo.
- **Metodo HTTP**: POST
- **Endpoint**: /group/join
- **Input richiesto**:
    - groupId (stringa): L'ID del gruppo a cui l'utente vuole unirsi.
- **Risposta**:
    - **Successo**: { message: 'User added to the group', group } con i dettagli del gruppo aggiornato.

- **Errore**:
  - { message: 'group not found' } se il gruppo non esiste.
  - { message: 'user is already in this group' } se l'utente è già membro del gruppo.
  - { message: 'cannot be in two groups simultaneously' } se l'utente è già membro di un altro gruppo.

3. *getUsers*

- **Descrizione**: Recupera gli utenti di un gruppo a cui l'utente autenticato è associato. Viene popolato il campo users per ottenere i dettagli degli utenti.
- **Metodo HTTP**: GET
- **Endpoint**: /group/users
- **Risposta**:
  - **Successo**: { message: 'users found', usersFound } con la lista degli utenti del gruppo.
  - **Errore**:
    - { message: 'users not found' } se non sono stati trovati utenti nel gruppo.

chatController.js

Il controller chatController.js gestisce le operazioni relative alla memorizzazione e al recupero dei messaggi di chat. Le principali operazioni comprendono l'invio di un messaggio in un gruppo e la lettura dei messaggi di un gruppo.

Funzioni del Controller

1. *saveMessage*

- **Descrizione**: Salva un nuovo messaggio di chat per un gruppo specifico. Il messaggio viene associato a un groupId e include il testo del messaggio, l'utente che lo ha inviato, e un timestamp.
- **Metodo HTTP**: POST
- **Endpoint**: /chat/message
- **Input richiesto**:
  - groupId (stringa): L'ID del gruppo a cui appartiene il messaggio.
  - messageText (stringa): Il testo del messaggio da inviare.
- **Risposta**:

- **Successo**: { message: 'message saved', newMessage } con i dettagli del messaggio appena creato.
- **Errore**:
  - { message: 'Error saving message' } se si verifica un errore durante il salvataggio del messaggio.

2. *getMessages*

- **Descrizione**: Recupera tutti i messaggi di un gruppo specifico, ordinati per timestamp in ordine crescente.
- **Metodo HTTP**: GET
- **Endpoint**: /chat/messages/:groupId
- **Input richiesto**:
  - groupId (stringa): L'ID del gruppo per cui si vogliono ottenere i messaggi.
- **Risposta**:
  - **Successo**: Un array di messaggi ordinati { messages }.
  - **Errore**:
    - { message: 'Error fetching messages' } se si verifica un errore durante il recupero dei messaggi.