

# Documentazione progetto React Keeper App

Francesco Merighi - VR474954  
Università degli Studi di Verona - A.A. 2023-2024

## SOMMARIO

<b>INTRODUZIONE</b>	<b>3</b>
Struttura del progetto	4
<b>FRONT-END</b>	<b>5</b>
React	5
Componenti utilizzati	5
App.jsx, index.js e index.html	5
Login.jsx e Signup.jsx	5
AuthForm.jsx	6
Home.jsx	8
Dashboard.jsx	9
Note.jsx	11
NoteForm.jsx	12
Header.jsx e Footer.jsx	14
Sidebar.jsx	16
NotFound.jsx	16
SessionExp.jsx	17
React Router e SPA	17
Implementazione: BrowserRouter, Routes e Route	17
Routes definite	18
Navigazione: Link e useNavigate()	18
Richieste HTTP con Axios	19
Async/await	19
File client-utils.js	19
Design e stile	23
CSS (Cascading Style Sheets)	23
Bootstrap	24
Material-UI	24
<b>BACK-END</b>	<b>25</b>
Node.js e NPM	25
Comandi utilizzati	25
Directory node_modules	25
File package.json	25
Express.js	26
Configurazione server	26

Configurazione di base	26
Dotenv (.env)	27
CORS	27
Middlewares	28
Avvio del server	29
Strategie di autenticazione	30
Passport.js e sessioni	30
Strategia locale	30
Strategia JWT	32
Endpoints definiti	33
File server-utils.js	37
<b>DATABASE</b>	<b>40</b>
PostgreSQL e pgAdmin	40
Tabelle	41
Connessione al database	41
Creazione istanza db	42
Connessione	42
Esecuzione query	43
Hashing delle password	43
Creazione hash	43
Comparazione password	44
<b>TESTING e VERSIONING</b>	<b>45</b>
Test sviluppatore	45
Test endpoints con Postman	45
Test utente	46
GitHub	46
<b>POSSIBILI ESTENSIONI FUTURE</b>	<b>48</b>
OAuth con Google	48
Più categorie di note	48
Switch modalità chiara-scura	48

# INTRODUZIONE

Il progetto si concentra sulla realizzazione di un'applicazione web che consente agli utenti di creare, visualizzare, modificare ed eliminare note personali in modo semplice e intuitivo.

L'applicazione è stata sviluppata utilizzando una tecnologia Full-Stack composta da:

- **FRONT-END**
  - REACT → libreria JavaScript per la creazione di interfacce utente dinamiche e reattive
  - BOOTSTRAP → framework che fornisce una vasta gamma di componenti e stili predefiniti
  - MATERIAL UI → libreria di componenti React che implementa il Material Design di Google
- **BACK-END**
  - NODE.JS → utilizzato per gestire lo sviluppo lato server, le dipendenze dell'app, i pacchetti da installare, ecc...
  - EXPRESS.JS → framework web per Node.js che semplifica la creazione di server e API web
- **DATABASE**
  - POSTGRESQL → database SQL relazionale (RDBMS) per la persistenza dei dati

Una volta registrati, gli utenti possono accedere alla loro dashboard personale dove possono gestire le proprie note personali.

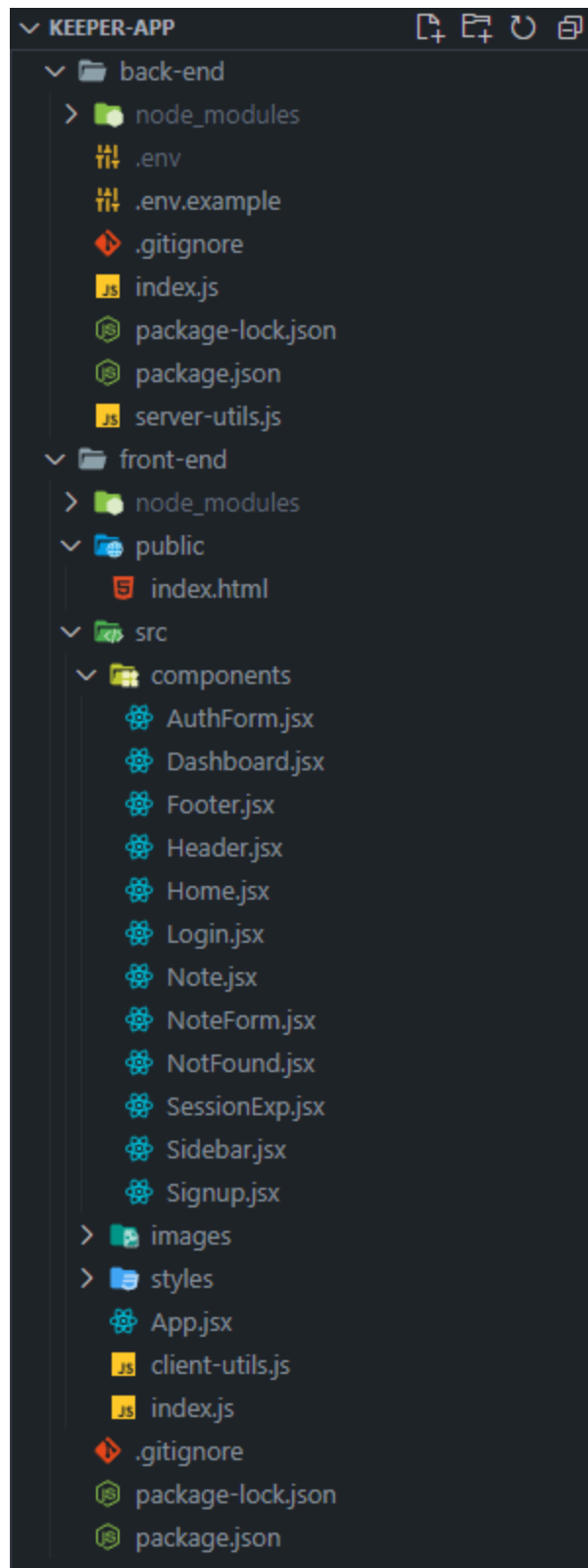
All'interno della dashboard, gli utenti possono organizzare le proprie note in due categorie distinte: "Principale" e "Lavoro".

Nella sezione "Principale", gli utenti possono decidere di gestire annotazioni di carattere generale.

Nella sezione "Lavoro", gli utenti possono decidere di gestire annotazioni di carattere lavorativo, come ad esempio note legate ad un progetto, ad un incontro o ad eventi importanti.

Questo consente agli utenti di mantenere un'organizzazione chiara e una visione strutturata delle loro note.

## Struttura del progetto



## FRONT-END

### React

**REACT** è una libreria JavaScript che permette la creazione di interfacce utente dinamiche e interattive.

Le caratteristiche principali di React includono:

- Segue un approccio basato sui COMPONENTI, dove l'interfaccia utente viene divisa in piccoli componenti indipendenti, che possono essere riutilizzati
- Introduce JSX, una sintassi che consente di scrivere HTML all'interno di JavaScript
- Gestisce lo STATO dell'applicazione consentendo di mantenere e aggiornare dinamicamente i dati all'interno dell'applicazione
- I dati possono essere passati da un componente ad un altro come PROPRIETÀ (props)

### Componenti utilizzati

I **COMPONENTI** utilizzati per lo sviluppo di questa applicazione sono i seguenti:

#### *App.jsx, index.js e index.html*

Il componente **App.jsx** rappresenta il contenitore principale dell'applicazione e contiene altri componenti al suo interno, definendo così la struttura dell'interfaccia utente:

- Al suo interno viene utilizzato [React Router](#) per definire le varie routes per quanto riguarda il Front-End

Il file **index.js** è il punto di partenza dell'applicazione e viene utilizzato per renderizzare il componente principale **App.jsx** all'interno del DOM (Document Object Model):

- Utilizza la funzione `ReactDOM.createRoot()` per creare un elemento "root" all'interno del `<div>` contenuto nel file *index.html* con id "root"
- Una volta creato l'elemento root, utilizza il metodo `render()` per renderizzare il componente **App.jsx** all'interno di esso

Nel file **index.html** viene definito un div con un id specifico "root" in cui l'applicazione React verrà renderizzata.

#### *Login.jsx e Signup.jsx*

Questi due componenti rappresentano rispettivamente le pagine di **LOGIN (accesso)** e di **SIGNUP (registrazione)**:

Al loro interno viene importato ed utilizzato il componente *AuthForm.jsx*, a cui viene passata una prop chiamata *“type”* che identifica il form da visualizzare:

- Nel componente *Signup.jsx* → *type* equivale a *“signup”*
- Nel componente *Login.jsx* → *type* equivale a *“login”*

### *AuthForm.jsx*

Questo componente rappresenta il **FORM** da visualizzare nel momento in cui l'utente vuole registrarsi/accedere.

Come detto in precedenza, questo componente accetta una prop chiamata *“type”*, che può essere di due tipi: *“signup”* o *“login”*.

- Se *props.type === “signup”* → gli *<input>* del form visualizzati saranno quelli relativi ad **email, username, password e conferma della password**
- Se *props.type === “login”* → gli *<input>* del form visualizzati saranno quelli relativi ad **email e password**

Le variabili e gli stati presenti all'interno del componente sono i seguenti:

- *const navigate = useNavigate()* → funzione hook fornita da [React Router](#) utilizzata per spostarsi tra le diverse pagine dell'applicazione in risposta a determinati eventi o azioni dell'utente
  - Dopo essersi autenticato, l'utente viene portato alla dashboard con *navigate(“/dashboard”)*
  - Dopo essersi registrato, l'utente viene portato alla pagina di login per procedere all'autenticazione con *navigate(“/login”)*
- *const baseUrl = ‘...’* → contiene l'URL del server *Express* a cui il client eseguirà le richieste HTTP
- *const [formData, setFormData] = useState({})* → variabile di stato rappresentata da un oggetto che contiene l'input inserito dall'utente nel form
  - L'oggetto è composto dai seguenti campi (inizialmente vuoti): *email, username, password e confirmPassword*
- *const [error, setError] = useState(“”)* → variabile di stato rappresentata da una stringa (inizialmente vuota) che contiene eventuali errori restituiti dal server in fase di registrazione/accesso
  - L'errore viene mostrato in un apposito alert per avvertire l'utente

Le funzioni utilizzate all'interno di questo componente sono le seguenti:

- `function handleChange(event)` → funzione che aggiorna la variabile di stato `formData` ad ogni inserimento da parte dell'utente nei vari campi del form
  - Assegnata all'evento `onChange` di ogni input presente nel form
- `async function handleSubmit(event)` → funzione che gestisce la logica di invio dei dati, inseriti dall'utente nel form, al server
  - Per prima cosa, viene prevenuto il normale comportamento del browser al momento dell'invio dei dati con `event.preventDefault()`
  - Poi, all'interno di un blocco `try-catch` vengono eseguite le richieste al server, in base al valore della prop `type`:
    - Se `props.type === "signup"` → richiesta POST contenente l'oggetto `formData` all'endpoint `/api/signup`

```
response = await axios.post('http://localhost:5000/api/signup', formData);
```

- Se la richiesta va a buon fine, l'utente viene registrato nel database e portato alla pagina di accesso per procedere con l'autenticazione
  - Se la richiesta non va a buon fine, viene mostrato un alert con il relativo errore
- Se `props.type === "login"` → richiesta POST contenente l'oggetto `formData` all'endpoint `/api/login`

```
response = await axios.post('http://localhost:5000/api/login', formData);
```

- Se la richiesta va a buon fine, l'utente viene autenticato e portato alla dashboard, poi viene salvato il `token JWT` contenuto nella risposta del server nel `localStorage`
  - Se la richiesta non va a buon fine, ad esempio se l'utente ha inserito delle credenziali non corrette, viene mostrato un alert con il relativo errore
- Infine, vengono resettati i campi del form
  - Assegnata all'evento `onSubmit` del form

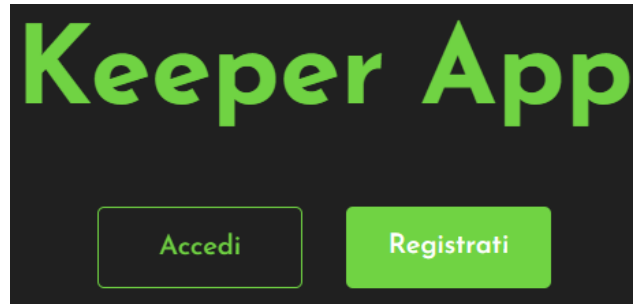
## Home.jsx

Questo componente rappresenta la prima pagina (ovvero la pagina **HOME**) che viene mostrata all'utente quando visita l'applicazione web.

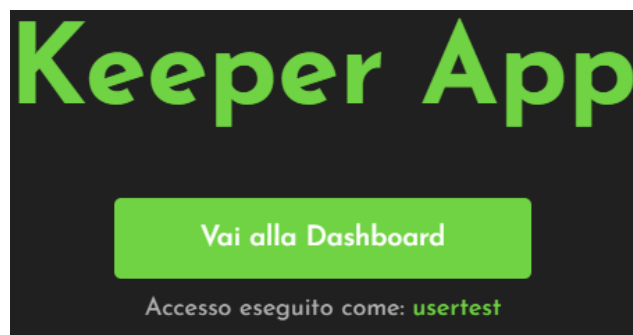


Questa pagina contiene degli elementi che vengono renderizzati dinamicamente in base al valore della variabile di stato `isTokenValid`

- Se `isTokenValid` è `false` → utente NON autenticato, vengono mostrati i pulsanti “ACCEDI” e “REGISTRATI”



- Se `isTokenValid` è `true` → utente autenticato, viene mostrato un pulsante “VAI ALLA DASHBOARD”, indicando l’username dell’utente autenticato



Le variabili e gli stati presenti all’interno di questo componente sono i seguenti:

- `const token = localStorage.getItem('token')` → se esiste, viene salvato il valore del Token JWT contenuto nel `localStorage`
- `const [isTokenValid, setIsTokenValid] = useState(null)` → variabile di stato rappresentata da un valore booleano (inizialmente `null`) che indica se il Token JWT è valido (utente autenticato), oppure no
- `const [user, setUser] = useState(null)` → variabile di stato che rappresenta un oggetto contenente le informazioni dell’utente autenticato
  - Viene poi mostrato l’username dell’utente con `user.username`
- `const [loading, setLoading] = useState(false)` → variabile di stato rappresentata da un valore booleano (inizialmente `false`) che indica lo stato di caricamento/aggiornamento dei dati
  - Se `loading === true` → viene mostrato uno spinner di caricamento
  - Se `loading === false` → viene mostrata la pagina con i dati aggiornati

---

Per aggiornare i valori delle variabili di stato *isTokenValid*, *user* e *loading*, vengono utilizzate due funzioni che, per comodità, sono state create in un file esterno chiamato [client-utils.js](#) (spiegate nel dettaglio nel relativo capitolo):

- `checkToken(token, setIsTokenValid, setLoading)` → tramite una richiesta al server, controlla se il token è valido e aggiorna la variabile *isTokenValid* di conseguenza
- `getUser(token, setUser, setLoading)` → tramite una richiesta al server, aggiorna la variabile *user* con le informazioni dell'utente autenticato

Queste due funzioni, vengono specificate all'interno di un hook React, chiamato *useEffect()*.

Questo hook, accetta come secondo parametro un array di dipendenze, che specifica quando il codice contenuto al suo interno deve essere eseguito.

Nel caso di questo componente, l'array di dipendenze specificato è *[token]*, per cui, le due funzioni sopra citate vengono eseguite OGNI volta che il valore di *token* cambia.

## *Dashboard.jsx*

Questo componente rappresenta la **DASHBOARD** che viene mostrata all'utente dopo aver eseguito l'accesso ed essere stato autenticato.

Al momento, l'applicazione permette ad un utente di creare note in due categorie distinte:

- Annotazioni di carattere generale → sezione "Principale" della dashboard
- Annotazioni di carattere lavorativo → sezione "Lavoro" della dashboard

Per differenziare queste due categorie, viene passata una prop chiamata *"category"* al componente, che può assumere come valore la stringa *"principale"* o la stringa *"lavoro"*.

Questo permette di utilizzare un solo componente *Dashboard*, che si comporti adeguatamente in base alla categoria, piuttosto che due componenti *Dashboard* distinti.

L'utente, ha la possibilità di creare una nota, premendo sul pulsante + o modificare/eliminare una nota, premendo i relativi pulsanti sulla nota stessa

Le variabili e gli stati presenti all'interno di questo componente sono i seguenti:

- `const token = localStorage.getItem("token")` → se esiste, viene salvato il valore del [Token JWT](#) contenuto nel localStorage

- `const [isTokenValid, setIsTokenValid] = useState(null)` → variabile di stato rappresentata da un valore booleano (inizialmente `null`) che indica se il Token JWT è valido (utente autenticato), oppure no
  - Se `isTokenValid === true` → viene mostrata la dashboard
  - Se `isTokenValid === false` → viene mostrato il componente [SessionExp](#)
- `const [notes, setNotes] = useState([])` → variabile di stato rappresentata da un array (inizialmente vuoto) che contiene le note scritte dall'utente
- `const [loading, setLoading] = useState(false)` → variabile di stato rappresentata da un valore booleano (inizialmente `false`) che indica lo stato di caricamento/aggiornamento dei dati
  - Se `loading === true` → viene mostrato uno spinner di caricamento
  - Se `loading === false` → viene mostrata la pagina con i dati aggiornati
- `const [isCreateActive, setIsCreateActive] = useState(false)` → variabile di stato rappresentata da un valore booleano (inizialmente `false`) che indica la visibilità/invisibilità del form di creazione di una nota
  - Se `isCreateActive === true` → l'utente ha premuto su +, form visibile
  - Se `isCreateActive === false` → form non più visibile
- `const [successMessage, setSuccessMessage] = useState("")` → variabile di stato rappresentata da una stringa che conterrà al suo interno i vari messaggi di successo mostrati nella dashboard
  - Se `successMessage` è una stringa vuota → non viene mostrato alcun alert
  - Se `successMessage` contiene una stringa
    - Viene mostrato un alert contenente il messaggio
    - L'alert viene mostrato per 5 secondi, poi scompare

Le funzioni utilizzate all'interno di questo componente sono le seguenti:

- `function handleCreateClick()` → imposta lo stato della variabile `isCreateActive` a `true`
  - Assegnata all'evento `onClick` del pulsante di aggiunta (+)

Il controllo di validità del token e il recupero delle note associate all'utente per categoria, vengono eseguiti chiamando le funzioni contenute nel file [client-utils.js](#):

- `checkToken(token, setIsTokenValid, setLoading)` → tramite una richiesta al server, controlla se il token è valido e aggiorna la variabile `isTokenValid` di conseguenza
- `getNotes(token, setNotes, setLoading, category)` → tramite una richiesta al server, aggiorna la variabile `notes` con le note associate all'utente autenticato, in base alla categoria passata come parametro

Per visualizzare le note salvate nell'array *notes*, viene utilizzato il metodo *map()* disponibile sugli array, che permette di iterare ed eseguire un'azione su OGNI elemento dell'array.

Nel caso di questa applicazione, si itera con *map()* sull'array *notes* e per ogni elemento contenuto nell'array si crea un componente [Note](#), a cui vengono passate diverse prop.

In React, è consigliabile associare una prop chiamata *key* ad ogni elemento generato dinamicamente tramite il metodo *map()*.

Questo perché React utilizza le chiavi (*keys*) per identificare in modo efficiente e gestire gli elementi della lista durante il processo di rendering e aggiornamento.

Se non è presente alcuna nota nell'array *notes*, ovvero *notes.length < 0*, viene mostrato un messaggio che avvisa l'utente e lo invita a crearne una.

Dopo che l'utente provvede a creare/modificare/eliminare una nota, viene mostrato un messaggio di successo, che fa capire all'utente che l'operazione è andata a buon fine.

### [Note.jsx](#)

Questo componente rappresenta la singola **NOTA** visualizzata nella dashboard dopo essere stata creata.

La nota presenta un titolo, un contenuto e due pulsanti che permettono la modifica della nota o l'eliminazione.

I pulsanti di modifica ed eliminazione, vengono mostrati solamente se l'utente esegue l'hover (passa con il mouse) sulla nota in questione.

Le variabili e gli stati presenti all'interno di questo componente sono i seguenti:

- Il componente accetta le seguenti props:
  - *title* → il titolo della nota
  - *content* → il contenuto della nota
  - *id* → id della nota
  - *setNotes* → imposta la variabile di stato *notes*, utilizzato nella logica di eliminazione/modifica di una nota
  - *setSuccessMessage* → imposta la variabile di stato *successMessage*, utilizzato per mostrare/nascondere i messaggi di successo
  - *setLoading* → imposta la variabile di stato *loading*
  - *category* → contiene la categoria di note a cui si fa riferimento
- *const token = localStorage.getItem('token')* → se esiste, viene salvato il valore del [Token JWT](#) contenuto nel localStorage

- `const [isEditActive, setIsEditActive] = useState(false)` → variabile di stato rappresentata da un valore booleano (inizialmente `false`) che indica la visibilità/invisibilità del form di modifica di una nota
  - Se `isEditActive === true` → l'utente ha premuto sul pulsante di modifica, form visibile
  - Se `isEditActive === false` → form di modifica non più visibile
- `const [isHovered, setIsHovered] = useState(false)` → variabile di stato rappresentata da un valore booleano (inizialmente `false`) che indica se l'utente sta eseguendo l'hover su di una nota oppure no
  - Se `isHovered === true` → l'utente sta passando il mouse sulla nota, vengono mostrati i pulsanti di eliminazione/modifica
  - Se `isHovered === false` → vengono nascosti i pulsanti di eliminazione/modifica

Le funzioni utilizzate all'interno di questo componente sono le seguenti:

- `function handleMouseOver()` → imposta lo stato della variabile `isHovered` a `true`
  - Assegnata all'evento `onMouseOver` dell'elemento nota
- `function handleMouseOut()` → imposta lo stato della variabile `isHovered` a `false`
  - Assegnata all'evento `onMouseOut` dell'elemento nota
- `function handleEditClick()` → imposta lo stato della variabile `isEditActive` all'opposto del valore attuale
  - Assegnata all'evento `onClick` del pulsante di modifica, contenuto nella nota
- `function handleDeleteClick()` → gestisce l'eliminazione di una nota
  - Assegnata all'evento `onClick` del pulsante di eliminazione, contenuto nella nota
  - Al suo interno viene chiamata la funzione `deleteNote()` dichiarata nel file [client-utils.js](#)

## NoteForm.jsx

Questo componente rappresenta il **FORM** da mostrare nel momento in cui l'utente vuole creare/modificare una nota.

La struttura del form è la stessa, sia per la creazione, che per la modifica:

- Un elemento `<input>` per quanto riguarda il titolo della nota
- Un elemento `<textarea>` per quanto riguarda il contenuto della nota

L'unica cosa che cambia è il tipo di richiesta eseguita al server al momento dell'invio dei dati.

Per indicare al componente che si sta trattando di una modifica, gli viene passata una prop chiamata *method* che contiene il valore *"update"*.

Le variabili e gli stati presenti all'interno di questo componente sono i seguenti:

- Il componente, oltre a *method*, accetta anche altre props:
  - *setIsActive* → utilizzato per impostare *isCreateActive/isEditActive* a *false* quando si preme sul pulsante di chiusura del form (X)
  - *title (solo modifica)* → contiene il titolo della nota che l'utente vuole modificare
  - *id (solo modifica)* → contiene l'id della nota che l'utente vuole modificare, utilizzato nelle richieste al server
  - *category* → contiene la categoria di note a cui si fa riferimento
  - *setSuccessMessage* → imposta la variabile di stato *successMessage*, utilizzato per mostrare/nascondere i messaggi di successo
  - *setNotes* → imposta la variabile di stato *notes*, utilizzato nella logica di eliminazione/modifica di una nota
  - *setLoading* → imposta la variabile di stato *loading*
- *const token = localStorage.getItem("token")* → se esiste, viene salvato il valore del Token JWT contenuto nel *localStorage*
- *const [note, setNote] = useState()* → variabile di stato rappresentata da un oggetto che contiene l'input inserito dall'utente nel form
  - L'oggetto è composto dai seguenti campi: *titolo*, *contenuto* e *category*, ovvero la categoria di nota

Le funzioni utilizzate all'interno di questo componente sono le seguenti:

- *function handleCloseClick()* → funzione che modifica il valore della variabile di stato *isCreateActive/isEditActive*, impostandola su *false*
  - Assegnata all'evento *onClick* del pulsante di chiusura del form (X)
- *function handleChange(event)* → funzione che aggiorna la variabile di stato *note* ad ogni inserimento da parte dell'utente nei vari campi del form
  - Assegnata all'evento *onChange* di ogni input presente nel form
- *function handleCreate(event)* → funzione che gestisce la logica di invio dei dati per creare una nota, inseriti dall'utente nel form, al server
  - Per prima cosa, viene prevenuto il normale comportamento del browser al momento dell'invio dei dati con *event.preventDefault()*

- Viene chiamata la funzione `createNote(token, note, setIsActive)`, dichiarata nel file `client-utils.js`, che al suo interno esegue una richiesta al server, per creare una nota, dato l'oggetto `note`
- Assegnata all'evento `onSubmit` del form
- `function handleEdit(event)` → funzione che gestisce la logica di invio dei dati per modificare una nota, inseriti dall'utente nel form, al server
  - Per prima cosa, viene prevenuto il normale comportamento del browser al momento dell'invio dei dati con `event.preventDefault()`
  - Viene chiamata la funzione `editNote(token, id, note, setIsActive)`, dichiarata nel file `client-utils.js`, che al suo interno esegue una richiesta al server, per modificare una nota, dato l'oggetto `note` e il valore `id`
  - Assegnata all'evento `onSubmit` del form, solo se è presente la prop `method` con valore `"update"`

### Header.jsx e Footer.jsx

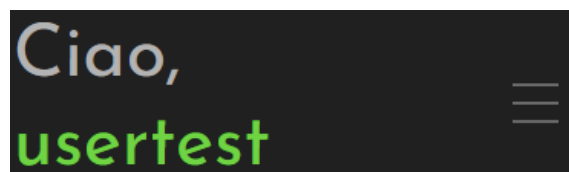
Il componente **HEADER** viene visualizzato quando l'utente si trova nella *Dashboard* e al suo interno vengono definiti alcuni elementi come:

- Un heading di benvenuto, che mostra l'username dell'utente autenticato
- Due pulsanti "Principale" e "Lavoro" che permettono di cambiare sezione della dashboard e mostrare le relative note
- Un pulsante "Esci" che permette all'utente di eseguire il logout e ritornare alla *Home*



Quando l'utente accede all'applicazione su dispositivi mobili, i 3 pulsanti citati in precedenza, vengono nascosti e al loro posto viene visualizzato un menù "hamburger".

Al tap su questo menù "hamburger", viene attivato e visualizzato il componente [Sidebar](#).



Le variabili e gli stati presenti all'interno di questo componente sono i seguenti:

- `const token = localStorage.getItem('token')` → se esiste, viene salvato il valore del [Token JWT](#) contenuto nel `localStorage`
- `const [user, setUser] = useState(null)` → variabile di stato che rappresenta un oggetto contenente le informazioni dell'utente autenticato
  - Viene poi mostrato l'username dell'utente con `user.username`
- `const [loading, setLoading] = useState(false)` → variabile di stato rappresentata da un valore booleano (inizialmente `false`) che indica lo stato di caricamento/aggiornamento dei dati
  - Se `loading === true` → viene mostrato uno spinner di caricamento
  - Se `loading === false` → viene mostrata la pagina con i dati aggiornati
- `const [isMenuActive, setIsMenuActive] = useState(false)` → variabile di stato rappresentata da un valore booleano (inizialmente `false`) che gestisce l'apertura/chiusura del menù "hamburger"
  - Se `isMenuActive === true` → viene mostrato il componente [Sidebar](#)
  - Se `isMenuActive === false` → il menù è chiuso
- Il componente accetta come props il metodo `setIsTokenValid`, utilizzato per impostare la variabile `isTokenValid` a `false` al momento del logout e `category` ovvero la categoria di note a cui si fa riferimento

Le funzioni utilizzate all'interno di questo componente sono le seguenti:

- `function handleLogoutClick()` → funzione che gestisce la logica di logout di un utente
  - Viene chiamata la funzione `logout(token, setLoading)`, dichiarata nel file [client.utils.js](#) che, tramite una richiesta al server, esegue la disconnessione dell'utente
  - Viene impostata la variabile di stato `isTokenValid` a `false`
  - Assegnata all'evento `onClick` del pulsante di logout
- `function handleMenuClick()` → funzione che aggiorna il valore della variabile di stato `isMenuActive`
  - Ad ogni click, la variabile di stato viene aggiornata con l'opposto del suo valore attuale
  - Assegnata all'evento `onClick` del menù "hamburger"

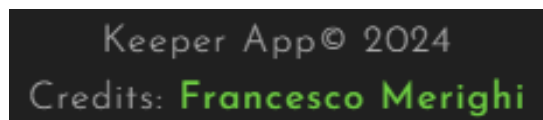
L'aggiornamento della variabile `user`, contenente le informazioni sull'utente autenticato, viene eseguito dalla funzione `getUser(token, setUser, setLoading)` (dichiarata all'interno del file [client-utils.js](#)), chiamata all'interno di un hook React `useEffect()`.



L'hook `useEffect()` specifica come array di dipendenze `[token]`, questo significa che la funzione sopra citata, viene eseguita ogni volta che `token` cambia valore.

Il componente **FOOTER** viene visualizzato solamente quando l'utente si trova nella *Home* e contiene alcune informazioni come:

- Nome dell'app, copyright © e anno, quest'ultimo generato dinamicamente grazie al metodo `getFullYear()` dell'oggetto `Date`
- Crediti di sviluppo



## *Sidebar.jsx*

Questo componente rappresenta la **SIDEBAR** che viene mostrata al click sul menù “hamburger”, quando l'app viene visualizzata su dispositivi mobili.

Esso contiene gli stessi link e pulsanti contenuti nell'*Header* ma disposti verticalmente per permettere una visione coerente su dispositivi mobili.

Le variabili e gli stati presenti all'interno di questo componente sono i seguenti:

- `const token = localStorage.getItem("token")` → se esiste, viene salvato il valore del Token JWT contenuto nel `localStorage`
- `const [loading, setLoading] = useState(false)` → variabile di stato rappresentata da un valore booleano (inizialmente `false`) che indica lo stato di caricamento/aggiornamento dei dati
  - Se `loading === true` → viene mostrato uno spinner di caricamento
  - Se `loading === false` → viene mostrata la pagina con i dati aggiornati

Le funzioni utilizzate all'interno di questo componente sono le seguenti:

- `function handleLogoutClick()` → funzione che gestisce la logica di logout di un utente
  - Viene chiamata la funzione `logout(token, setLoading)`, dichiarata nel file `client.utils.js` che, tramite una richiesta al server, esegue la disconnessione dell'utente
  - Viene impostata la variabile di stato `isTokenValid` a `false`
  - Assegnata all'evento `onClick` del pulsante di logout

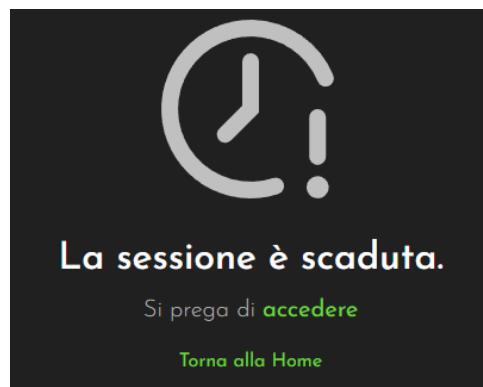
## *NotFound.jsx*

Questo componente rappresenta la pagina che viene mostrata all'utente quando tenta di accedere ad una route **NON SPECIFICATA**.



## *SessionExp.jsx*

Questo componente rappresenta la pagina che viene mostrata all'utente quando esso tenta di accedere alla dashboard senza essersi prima autenticato, oppure quando la **SESSIONE** è **SCADUTA**.



## React Router e SPA

Per far sì che l'utente riesca a navigare correttamente tra le pagine web che compongono l'applicazione, viene introdotto il concetto di **ROUTING**.

Il routing permette di definire una serie di "rotte" o "percorsi" (routes) all'interno dell'applicazione, associando a ciascuna di esse un URL specifico e il componente corrispondente da visualizzare quando l'utente visita quel particolare URL.

Esso è particolarmente importante nelle **SINGLE PAGE APPLICATIONS (SPA)** in cui l'interazione dell'utente avviene senza che la pagina venga ricaricata completamente dal server.

Il routing viene gestito lato client utilizzando una libreria o un framework specifico, come ad esempio **REACT ROUTER**.

### Implementazione: *BrowserRouter, Routes e Route*

L'implementazione del Routing per questa applicazione avviene nel componente principale *App.jsx*, grazie all'utilizzo di alcuni componenti forniti dalla libreria *React Router*:

- **BrowserRouter as Router** → è un componente wrapper utilizzato per fornire il contesto del Routing a tutta l'applicazione
  - consente di aggiornare dinamicamente l'URL del percorso del browser senza ricaricare la pagina
- **Routes** → è un componente dove al suo interno vengono definite le rotte dell'applicazione
  - può contenere una serie di componenti *Route* che rappresentano le singole rotte dell'applicazione
- **Route** → componente utilizzato per definire una singola rotta nell'applicazione e accetta diverse props
  - *path* → definisce l'URL della rotta
  - *element* → componente da renderizzare quando l'URL corrisponde al *path* definito

### *Routes definite*

Le **ROUTES** definite per questa applicazione sono le seguenti:

- **Route di login** → indirizza l'utente alla pagina di accesso
  - *path* → `/login`
  - *element* → `<Login />`, ovvero *Login.jsx*
- **Route di signup** → indirizza l'utente alla pagina di registrazione
  - *path* → `/signup`
  - *element* → `<Signup />`, ovvero *Signup.jsx*
- **Route di home** → indirizza l'utente alla pagina home
  - *path* → `/`
  - *element* → `<Home />` e `<Footer />`, ovvero *Home.jsx* e *Footer.jsx*

- **Route di *dashboard (principale)*** → indirizza l'utente alla dashboard (principale)
  - *path* → `/dashboard`
  - *element* → `<Dashboard category="principale" />`, ovvero `Dashboard.jsx`
- **Route di *dashboard (lavoro)*** → indirizza l'utente alla dashboard (lavoro)
  - *path* → `/dashboard/work`
  - *element* → `<Dashboard category="lavoro" />`, ovvero `Dashboard.jsx`
- **Route di *risorsa non trovata*** → indirizza l'utente alla pagina 404
  - *path* → `*`
  - *element* → `<NotFound />`, ovvero `NotFound.jsx`

### Navigazione: `Link` e `useNavigate()`

Per permettere all'utente di navigare tra le rotte dell'applicazione, viene utilizzato un componente fornito dalla libreria *React Router*, chiamato **Link**.

I vantaggi nell'utilizzare il componente `<Link />` sono i seguenti:

- Evita il caricamento completo della pagina quando l'utente fa clic sul link, rendendo l'applicazione più veloce e reattiva
- Consente di definire facilmente i link utilizzando l'attributo *to*, che specifica la rotta di destinazione del link

L'hook **`useNavigate`** invece viene utilizzato quando la navigazione deve essere gestita in risposta a eventi o azioni all'interno di un componente, nel caso dell'applicazione in questione:

- Al momento dell'accesso, l'utente viene indirizzato alla dashboard
- Al momento della registrazione, l'utente viene indirizzato alla pagina di accesso
- Al momento del logout, l'utente viene indirizzato alla home

### Richieste HTTP con Axios

Per eseguire le **RICHIESTE HTTP** ai vari endpoint specificati all'interno del server locale *Express.js* configurato, viene utilizzata una libreria molto intuitiva per quanto riguarda l'interazione con API esterne, l'invio e la ricezione di dati da server web e la gestione della comunicazione client-server, ovvero **Axios**.

Per mantenere un codice pulito e manutenibile è stato deciso di gestire le richieste HTTP eseguite dal client, in un file separato, chiamato [`client-utils.js`](#).

## Async/await

Axios restituisce una *Promise* quando viene eseguita una richiesta HTTP, ovvero il risultato di un'operazione asincrona.

Per gestire al meglio questo particolare oggetto, vengono utilizzate le keyword ***async/await***.

La keyword *async* anteposta alla dichiarazione di una funzione, indica che in quella funzione vengono eseguite una o più operazioni asincrone.

L'uso della keyword *await* su una *Promise* restituita da Axios, fa sì che l'esecuzione del codice attenda che la *Promise* si risolva (cioè che la richiesta HTTP venga completata) prima di procedere.

## File client-utils.js

All'interno di questo file, sono state create diverse funzioni, una per ogni richiesta effettuata da parte del client, che vengono esportate ed importate nei vari componenti dove e quando serve.

Inoltre, è stata dichiarata una variabile chiamata *baseURL* che contiene l'URL del server *Express* a cui il client eseguirà le richieste HTTP.

Per quanto riguarda la logica e le richieste effettuate al server al momento dell'accesso o della registrazione, esse vengono gestite direttamente nel componente [AuthForm.jsx](#)

Le funzioni dichiarate all'interno del file sono le seguenti:

- ***async function checkToken(token, setIsTokenValid, setLoading)***
  - Accetta tre parametri, il *token JWT*, la funzione setter per la variabile *isTokenValid* e la funzione setter per la variabile *loading*
  - Dentro al blocco *try-catch* viene eseguita una richiesta GET all'endpoint /api/validateToken e nell'header viene inserito il *token JWT*

```
const response = await axios.get('http://localhost:5000/api/validateToken', { headers: {  
  'Authorization': `Bearer ${token}`,  
}});
```

- Se la richiesta va a buon fine, ovvero il token è valido, allora viene impostata *isTokenValid* a *true*

- Se la richiesta non va a buon fine, ovvero il token non è valido, viene impostata *isTokenValid* a *false*
- Nella clausola *finally* viene impostato il valore della variabile *loading* a *false*, tramite il proprio metodo setter
- ***async function getUser(token, setUser, setLoading)***
  - Accetta tre parametri, il *token JWT*, la funzione setter per la variabile *user* e la funzione setter per la variabile *loading*
  - Dentro al blocco *try-catch* viene eseguita una richiesta GET all'endpoint */api/user* e nell'header viene inserito il *token JWT*

```
const response = await axios.get(`http://localhost:5000/api/user`, { headers: {  
  'Authorization': `Bearer ${token}`  
}});
```

- Se la richiesta va a buon fine, allora viene impostata *user* con l'oggetto *user*, associato al *token JWT*, contenuto nella risposta del server
- Se la richiesta non va a buon fine, ad esempio perché il *token JWT* non è valido, viene stampato un messaggio di errore
- Nella clausola *finally* viene impostato il valore della variabile *loading* a *false*, tramite il proprio metodo setter
- ***async function getNotes(token, setNotes, setLoading, category)***
  - Accetta quattro parametri, il *token JWT*, la funzione setter per la variabile *notes*, la funzione setter per la variabile *loading* e la categoria associata alle note che si vogliono recuperare
  - Dentro al blocco *try-catch* viene eseguita una richiesta GET all'endpoint */api/getNotes*, nell'header viene inserito il *token JWT* e nei *parametri query* viene inserita la categoria di note

```
const response = await axios.get(`http://localhost:5000/api/getNotes?category=${category}`, { headers: {  
  'Authorization': `Bearer ${token}`  
}});
```

- Se la richiesta va a buon fine, allora viene impostata *notes* con l'oggetto *notes*, contenuto nella risposta del server
- Se la richiesta non va a buon fine, ad esempio perché il *token JWT* non è valido, viene stampato un messaggio di errore
- Nella clausola *finally* viene impostato il valore della variabile *loading* a *false*, tramite il proprio metodo *setter*
- ***async function logout(token, setLoading)***
  - Accetta due parametri, il *token JWT*, e la funzione *setter* per la variabile *loading*
  - Dentro al blocco *try-catch* viene eseguita una richiesta GET all'endpoint /api/logout e nell'header viene inserito il *token JWT*

```
const response = await axios.get(`http://localhost:5000/api/logout`, { headers: {  
  'Authorization': `Bearer ${token}`  
}});
```

- Se la richiesta va a buon fine, allora il *token JWT* viene rimosso dal *localStorage* in quanto l'utente si è disconnesso
- Se la richiesta non va a buon fine, ad esempio perché il *token JWT* non è valido, viene stampato un messaggio di errore
- Nella clausola *finally* viene impostato il valore della variabile *loading* a *false*, tramite il proprio metodo *setter*
- ***async function deleteNote(token, id, setNotes, setSuccessMessage)***
  - Accetta tre parametri, il *token JWT*, l'id della nota che si vuole eliminare e la funzione *setter* per la variabile *notes*
  - Dentro al blocco *try-catch* viene eseguita una richiesta DELETE all'endpoint /api/deleteNote/\${id} e nell'header viene inserito il *token JWT*

```
const response = await axios.delete(`http://localhost:5000/api/deleteNote/${id}`, { headers: {  
  'Authorization': `Bearer ${token}`  
}});
```

- Se la richiesta va a buon fine, allora, all'interno del metodo *setNotes*, viene applicata la funzione *filter()* per ritornare un array di note senza la nota eliminata

- Se la richiesta non va a buon fine, ad esempio perché il *token JWT* non è valido, viene stampato un messaggio di errore

- ***async function editNote(token, id, note, setIsActive, setLoading, category)***

- Accetta quattro parametri, il *token JWT*, l'id della nota che si vuole modificare, l'oggetto *note* che rappresenta la nota modificata e la funzione setter per la variabile *isActive*
- Dentro al blocco *try-catch* viene eseguita una richiesta PUT all'endpoint /api/editNote/\${id}, nell'header viene inserito il *token JWT* e nel body l'oggetto *note*

```
const response = await axios.put(`http://localhost:5000/api/editNote/${id}`, note, {
  headers: { Authorization: `Bearer ${token}` }
});
```

- Se la richiesta va a buon fine, la nota viene modificata, viene impostata la variabile *isActive* a *false* (il form non è più visibile) e la pagina viene ricaricata
- Se la richiesta non va a buon fine, ad esempio perché il *token JWT* non è valido, viene stampato un messaggio di errore

- ***async function createNote(token, note, setIsActive, setNotes, setLoading, setSuccessMessage, category)***

- Accetta tre parametri, il *token JWT*, l'oggetto *note* che rappresenta la nota creata e la funzione setter per la variabile *isActive*
- Dentro al blocco *try-catch* viene eseguita una richiesta POST all'endpoint /api/createNote, nell'header viene inserito il *token JWT* e nel body l'oggetto *note*

```
const response = await axios.put(`http://localhost:5000/api/editNote/${id}`, note, {
  headers: { Authorization: `Bearer ${token}` }
});
```

- Se la richiesta va a buon fine, la nota viene creata, viene impostata la variabile *isActive* a *false* (il form non è più visibile) e la pagina viene ricaricata
- Se la richiesta non va a buon fine, ad esempio perché il *token JWT* non è valido, viene stampato un messaggio di errore



---

## Design e stile

Per dare uno stile unico all'applicazione e creare interfacce che offrano all'utente la miglior esperienza possibile, sono state utilizzate diverse tecnologie.

### *CSS (Cascading Style Sheets)*

Grazie all'utilizzo dei fogli di stile **CSS**, è stato possibile definire il design e il layout principale dell'app.

Ogni componente ha un foglio di stile associato, che contiene le proprietà definite per gli elementi che si trovano al suo interno.

In particolare, in questi fogli di stile, viene gestito:

- Layout dell'applicazione utilizzando una combinazione di *Flexbox* e *Grid*
- Colori, tipografia, spaziatura ed altri elementi di design
- Responsività, ovvero l'adattamento dell'app in base alla grandezza del viewport, tramite l'utilizzo di *@media* queries
- Animazioni, definendo dei *@keyframes* per ogni tipo di animazione e utilizzandoli dove serve

### *Bootstrap*


**BOOTSTRAP** fornisce un set completo di componenti UI, stili predefiniti e strumenti di layout che semplificano il processo di creazione di interfacce utente moderne.

Sono stati utilizzati alcuni elementi forniti da questo framework:

- Viene utilizzato il componente `<Container />`, progettato per visualizzare il contenuto principale di una pagina web all'interno di un'area definita
- Viene utilizzato il componente `<Alert />`, per visualizzare un alert che mostra eventuali messaggi di errore
- Quando la pagina è in stato di caricamento, viene mostrato uno *spinner*, anch'esso fornito da Bootstrap

### *Material-UI*

**MATERIAL-UI** è una libreria di componenti UI React che implementa il design system Material Design sviluppato da Google.



Da questa libreria, sono state importate ed utilizzate alcune icone, come ad esempio, l'icona per l'aggiunta di una nota (+), l'icona di modifica di una nota e l'icona per l'eliminazione.

## BACK-END

### Node.js e NPM

**NPM (Node Package Manager)** è il gestore di pacchetti predefinito per Node.js.

Esso viene utilizzato per installare e gestire framework, librerie, plugin e strumenti, disponibili nel registro pubblico dei pacchetti di NPM, tramite una serie di comandi inseriti da terminale.

#### Comandi utilizzati

I comandi utilizzati per installare e gestire pacchetti e/o avviare l'applicazione, sono i seguenti:

- **`npx create-react-app keeper-app`** → utilizzato per creare ed inizializzare un nuovo progetto React, chiamato *keeper-app*
- **`npm i (o install) nome-pacchetto`** → utilizzato per installare un pacchetto identificato con *nome-pacchetto*, viene poi aggiunta una dipendenza relativa ad esso, nel file *package.json*
- **`npm i (o install)`** → utilizzato principalmente quando un progetto viene scaricato/clonato, installa tutte le dipendenze specificate nel file *package.json* necessarie per il corretto funzionamento dell'applicazione
- **`npm start`** → utilizzato per avviare l'applicazione (client) all'indirizzo <http://localhost:3000> (il numero di porta potrebbe cambiare se quest'ultima è occupata da un altro processo in esecuzione)
- **`nodemon server/server.js`** → utilizzato per avviare il server *Express.js* all'indirizzo <http://localhost:5000> (o in base al valore di *process.env.PORT*)

#### Directory *node\_modules*

La directory ***node\_modules*** viene generata automaticamente all'interno del progetto quando si installano pacchetti *npm*.

Questa cartella contiene tutti i pacchetti di codice JavaScript necessari al progetto, inclusi i pacchetti principali e le relative dipendenze.

#### File *package.json*

---

Il file ***package.json*** è un file di configurazione utilizzato nel progetto per definire le dipendenze, gli script di avvio e altre configurazioni pertinenti al progetto.

Quando si esegue il comando *npm i nome-pacchetto* per installare un pacchetto all'interno del progetto, esso viene automaticamente aggiunto alle dipendenze specificate nel file *package.json*

Quando, dopo aver scaricato/clonato il progetto, si esegue *npm i*, viene fatto riferimento a questo file, per determinare le dipendenze da installare necessarie per il corretto funzionamento dell'applicazione.

## Express.js

**EXPRESS.JS** è un framework web leggero e flessibile per Node.js, progettato per semplificare lo sviluppo di server web e gestire le richieste HTTP in modo efficiente.

Alcune delle funzionalità offerte da *Express.js* sono:

- Offre un sistema di routing che consente di definire facilmente gli ENDPOINTS e le relative azioni da eseguire per gestire le RICHIESTE HTTP
- Supporta una vasta gamma di MIDDLEWARES, ovvero funzioni intermedie che possono essere utilizzate per elaborare le richieste HTTP

## Configurazione server

Qui di seguito vengono spiegate le parti di codice che riguardano la configurazione del server *Express.js*.

Le configurazioni riguardanti l'istanza del database e la connessione ad esso, vengono spiegate nel relativo capitolo.

### Configurazione di base

Dopo aver installato *Express.js* tramite npm e dopo averlo importato correttamente, si procede alla creazione di un'istanza *app* utilizzando la funzione *express()*.

```
const app = express();
```

Questo oggetto *app* rappresenta l'applicazione web e sarà utilizzato per definire gli endpoints, i middlewares e altre configurazioni.

Oltre all'oggetto *app* viene definita una costante *port* che contiene il numero di porta su cui il server verrà avviato.

```
const port = process.env.PORT || 5000;
```

In questo caso, `process.env.PORT` tenta di leggere il valore della variabile d'ambiente `PORT`, che potrebbe essere configurata in un apposito file di configurazione oppure “gestita” dal servizio di hosting utilizzato quando viene eseguito il deploy dell'applicazione.

Se la variabile d'ambiente `PORT` non è definita o non è leggibile, la parte “`|| 5000`” specifica che il server Express dovrebbe mettersi in ascolto sulla porta `5000` per le richieste HTTP in ingresso.

## Dotenv (.env)

Un file `.env` è un file di configurazione che contiene variabili d'ambiente che fanno riferimento ad informazioni sensibili come chiavi API, password di database e altre configurazioni specifiche dell'ambiente di sviluppo.

Le variabili d'ambiente definite per questa applicazione sono le seguenti:

- **DB\_PASSWORD** → password d'accesso al database PostgreSQL *keeper-app*
- **SESSION\_SECRET** → chiave segreta utilizzata nella configurazione delle sessioni
- **JWT\_SECRET** → chiave segreta utilizzata nella configurazione di JWT

Per facilitare la gestione di queste variabili d'ambiente all'interno dell'applicazione, viene utilizzata una libreria chiamata **dotenv**.

Essa permette di leggere le variabili d'ambiente definite in un file `.env` e di renderle disponibili come variabili di processo (accessibili con `process.env`) nell'applicazione, utilizzando la funzione `dotenv.config()`.

```
dotenv.config();
```

## CORS

Quando l'applicazione in esecuzione su <http://localhost:3000> tenta di fare richieste HTTP ad un server *Express.js* in esecuzione su <http://localhost:5000>, il browser riconosce che l'origine delle richieste è diversa e applica la politica di sicurezza **CORS**.

La politica CORS è un meccanismo di difesa cruciale per proteggere gli utenti web e l'applicazione da attacchi di sicurezza.

Per consentire al browser di effettuare richieste da client a server bisogna specificare l'origine all'interno della configurazione del server, utilizzando la libreria **cors**.

```
const corsOptions = {
  origin: ['http://localhost:3000'],
  optionsSuccessStatus: 200,
};
```

Per specificare l'origine viene definito un oggetto *corsOptions* contenente il campo *origin* che indica l'origine consentita.

## Middlewares

I **MIDDLEWARES** agiscono come uno strato intermedio tra le richieste HTTP in ingresso e la logica di gestione delle richieste effettiva dell'applicazione.

Essi possono, ad esempio:

- eseguire operazioni come il parsing dei dati delle richieste
- gestire il processo di autenticazione degli utenti
- registrare le richieste in ingresso e i relativi dettagli

Per utilizzare un middleware all'interno di un'applicazione, viene utilizzata la funzione *app.use()* fornita da *Express.js*.

I middlewares utilizzati per questa applicazione sono i seguenti:

- **CORS Middleware** → gestisce le politiche di sicurezza CORS, consentendo le richieste da origini specificate nell'oggetto *corsOptions*  

```
app.use(cors(corsOptions));
```
- **Body Parser Middlewares** → analizzano il corpo delle richieste in arrivo e popolano l'oggetto *req.body* con i dati inviati dal client (ad esempio tramite form)

```
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

- **Session Middleware** → gestisce le sessioni degli utenti, memorizzando e recuperando le informazioni di sessione dall'oggetto *req.session*
  - *secret* → chiave segreta contenuta nel file di configurazione *.env*
  - Gli altri due *parametri* specificano se le sessioni devono essere salvate anche se sono vuote o non sono state modificate durante la richiesta

```
app.use(session({
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
}));
```

- **Passport.js Middlewares** → gestiscono l'inizializzazione di *Passport.js* e l'integrazione di esso con l'utilizzo delle sessioni

```
app.use(passport.initialize());
app.use(passport.session());
```

- **Logger Middleware** → registra le richieste HTTP in ingresso e mostra i relativi dettagli nella console

```
app.use(logger);
```

La funzione *logger()*, creata ad hoc per fare ciò, è la seguente:

```
function logger(req, res, next) {
  console.log(`Richiesta ${req.method} per ${req.url}`);
  next();
}
```

Viene chiamata la funzione *next()* per passare la richiesta al middleware successivo nell'elenco o alla logica di gestione effettiva della richiesta.

```
Richiesta GET per /api/validateToken
```

## Avvio del server

Per **AVVIARE** il server viene utilizzata la funzione *app.listen()* fornita direttamente da *Express.js*, che accetta i seguenti parametri:

```
app.listen(port, '0.0.0.0', () => {
  console.log(`Server in ascolto sulla porta ${port}`);
});
```

- *port* → porta su cui il server si metterà in ascolto per le richieste HTTP
- *0.0.0.0* → il server è in ascolto su tutte le interfacce di rete disponibili
- *funzione di callback* che viene eseguita una volta che il server è stato avviato

## Strategie di autenticazione

### *Passport.js e sessioni*

**PASSPORT.JS** è un middleware molto utilizzato per gestire l'autenticazione degli utenti in un'applicazione web.

Esso supporta una vasta gamma di strategie di autenticazione, consentendo di implementare facilmente l'autenticazione tramite username/password, OAuth, JWT e altro ancora.

Passport.js utilizza le **SESSIONI** per memorizzare lo stato di autenticazione dell'utente e durante le richieste successive verificherà se le informazioni di autenticazione dell'utente (come il *Token JWT*) sono presenti nella sessione per determinare se l'utente è autenticato o meno.

### *Strategia locale*

Nell'applicazione, viene implementata tramite la libreria *passport-local*, una **STRATEGIA DI AUTENTICAZIONE LOCALE** che consente di autenticare gli utenti utilizzando credenziali (*username/password*) memorizzate all'interno del database.

Viene definita una strategia di autenticazione locale chiamata *"login"* utilizzando *passport.use()*.

Essa viene istanziata con *new LocalStrategy*, che accetta un oggetto di opzioni e una funzione di callback che verrà eseguita quando viene effettuato un tentativo di autenticazione.

Le opzioni della strategia specificano i campi del form di autenticazione HTML che devono essere utilizzati per l'email e la password, nel caso di questa applicazione:

- *usernameField* corrisponde ad *email*
- *passwordField* corrisponde a *password*

La *funzione di callback* accetta tre parametri, ovvero *email*, *password* e la funzione *done()*, fornita da *Passport.js*, utilizzata per segnalare il completamento dell'autenticazione o per gestire eventuali errori.

La funzione *done()* accetta principalmente tre argomenti:

1. *error* → indica se si è verificato un errore durante il processo di autenticazione
2. *user* → rappresenta l'utente autenticato
3. *info* → fornisce informazioni aggiuntive sull'esito dell'autenticazione

In base alle diverse situazioni che si possono presentare, la funzione *done()* assume dei valori coerenti.



- Per prima cosa, viene eseguita una ricerca dell'utente nel database utilizzando l'email fornita (*findUser()* è una funzione creata in un file esterno chiamato [server-utils.js](#))
- Se l'utente NON viene trovato, viene chiamata la funzione *done()* con i seguenti argomenti:
  - *null* → non si è verificato nessun errore durante il processo
  - *false* → nessun utente autenticato
  - *message* → informa che l'utente non è stato trovato
- Se l'utente viene trovato, viene chiamata la funzione *done()* con i seguenti argomenti:
  - *null* → non si è verificato nessun errore durante il processo
  - *user* → rappresenta l'utente autenticato
  - *message* → informa che l'utente è stato autenticato con successo
- Se l'utente viene trovato, ma le password non corrispondono, viene chiamata la funzione *done()* con i seguenti argomenti:
  - *null* → non si è verificato nessun errore durante il processo
  - *false* → nessun utente autenticato
  - *message* → informa che la password è errata

La password inserita viene confrontata con la password criptata salvata nel database, tramite la funzione *compare()* della libreria *bcrypt*.
- Se c'è un errore generico (ad esempio di connessione al server), viene chiamata la funzione *done()* con i seguenti argomenti:
  - *error* → oggetto che identifica l'errore
  - *false* → nessun utente autenticato
  - *message* → informa che qualcosa è andato storto

```
passport.use("login", new LocalStrategy({ usernameField: 'email', passwordField: 'password' }, async (email, password, done) => {
  try {
    const user = await findUser(email);

    if (!user) {
      return done(null, false, { message: 'Utente non trovato' });
    }

    const isPasswordValid = await bcrypt.compare(password, user.password);

    if (!isPasswordValid) {
      return done(null, false, { message: 'Password errata' });
    }

    return done(null, user, { message: 'Utente autenticato con successo' });
  } catch (error) {
    return done(error, false, { message: 'Qualcosa è andato storto' });
  }
}));
```

Questa strategia viene utilizzata quando viene eseguita una richiesta POST all'endpoint /api/login.

## Strategia JWT

Nell'applicazione, viene implementata tramite la libreria passport-jwt, una **STRATEGIA DI AUTENTICAZIONE JWT (JSON WEB TOKEN)** che permette di autenticare richieste utilizzando token firmati e criptati contenenti informazioni sull'utente.

Viene definita una strategia di autenticazione JWT, utilizzando *passport.use()*.

Essa viene istanziata con *new JWTStrategy*, che accetta un oggetto di configurazione e una funzione di verifica del token JWT, che verrà eseguita quando un token JWT viene ricevuto e deve essere verificato.

L'oggetto di configurazione specifica come e dove trovare il token JWT nelle richieste e la chiave segreta utilizzata per firmare e verificare i token:

- Il token JWT viene estratto dall'header "*Authorization*" delle richieste HTTP come *token Bearer*

La *funzione di callback* accetta due parametri, ovvero *jwtPayload* che contiene le informazioni decodificate dal token e la funzione *done()*, fornita da *Passport.js*, utilizzata per segnalare il completamento dell'autenticazione o per gestire eventuali errori.

Viene cercato l'utente nel database utilizzando l'ID dell'utente estratto dal token JWT, tramite la funzione *findUserById()* creata nel file esterno server-utils.js e in base al risultato, la funzione *done()* assume dei valori coerenti.

- Se l'utente viene trovato, indica a *Passport.js* che l'utente è autenticato e viene ritornata la funzione *done()* con i seguenti argomenti:
  - *null* → non si è verificato nessun errore durante il processo
  - *user* → rappresenta l'utente autenticato
- Se l'utente NON viene trovato o si verifica un errore durante il processo, viene ritornata la funzione *done()* con i seguenti argomenti:
  - *error* → oggetto che identifica l'errore
  - *false* → nessun utente autenticato

```

passport.use(new JWTStrategy.Strategy({
  jwtFromRequest: JWTStrategy.ExtractJwt.fromAuthHeaderAsBearerToken(),
  secretOrKey: process.env.JWT_SECRET
}, async (jwtPayload, done) => {
  try {
    const user = await findUserById(jwtPayload.user.id);
    return done(null, user);
  } catch (error) {
    return done(error, false);
  }
}));

```

Questa strategia viene utilizzata OGNI volta che l'utente cerca di eseguire una richiesta HTTP ad una risorsa protetta.

## Endpoints definiti

Vengono definiti vari **ENDPOINTS** a cui il client può fare richiesta, alcuni sono “liberi”, altri sono “protetti” e richiedono un token JWT valido per far sì che la richiesta venga presa in carico.

- Endpoint POST per la REGISTRAZIONE → `/api/signup`

Per prima cosa, vengono presi i valori di *email*, *username*, *password* e *confirmPassword* dal *body* della richiesta.

Poi, vengono eseguiti vari controlli sui valori, come ad esempio, *password* deve essere di almeno 4 caratteri e deve combaciare con il valore di *confirmPassword*.

- Nel caso qualche campo NON superi i controlli, il server risponde con un messaggio di errore e relativo codice di stato (400)

Successivamente, all'interno di un blocco *try-catch*, viene controllato se esiste già un utente registrato, tramite la funzione *findUser()* dichiarata nel file [server-utils.js](#). Se l'utente esiste già, il server risponde con un messaggio di errore e relativo codice di stato (400)

Se l'utente non esiste, vengono eseguite le seguenti operazioni:

- Viene creato un *hash* della *password* utilizzando la funzione *bcrypt.hash()*
- Viene registrato l'utente nel database, tramite la funzione *createUser()* dichiarata nel file [server-utils.js](#).
- Il server risponde con un messaggio di successo e relativo codice di stato (200)

Se si verifica un errore generico (ad esempio di connessione al server), il server risponde con un messaggio di errore e relativo codice (500).

- Endpoint POST per l'ACCESSO → `/api/login`

Quando viene effettuata una richiesta a questo endpoint, viene utilizzata la STRATEGIA DI AUTENTICAZIONE LOCALE chiamata “login”.

La *funzione di callback* fornita come secondo argomento, viene chiamata con tre parametri:

- *error* → indica un errore che si verifica durante il processo di autenticazione
- *user* → oggetto che rappresenta l'utente autenticato
- *info* → oggetto contenente informazioni come messaggi di errore/successo

All'interno della funzione, viene eseguita la logica di accesso per un utente:

- Se si verifica un errore generico (ad esempio di connessione al server), il server risponde con un messaggio di errore e relativo codice (500).
- Se l'utente non viene trovato, il server risponde con un messaggio di errore che proviene dalle funzioni *done()* dichiarate nella strategia e relativo codice (400)
- Se l'autenticazione ha successo e l'utente viene trovato, viene effettuato il login utilizzando *req.login()*

Dopo il login riuscito, viene generato un *token JWT* tramite la funzione *sign()*, contenente le informazioni sull'utente e firmato utilizzando la relativa chiave segreta specificata nel file di configurazione *.env*.

Questo *token* ha una durata di validità di 30 minuti, questo significa che, passato questo lasso di tempo, la sessione scade e viene richiesto all'utente di autenticarsi nuovamente.

Infine, vengono restituiti dal server, un messaggio di successo, le informazioni sull'utente e il *token JWT* generato, che poi verrà salvato nel *localStorage* al momento dell'accesso nel FRONT-END.

- Endpoint POST per la CREAZIONE DI UNA NOTA → `/api/createNote`

Quando viene effettuata una richiesta a questo endpoint, viene utilizzata la STRATEGIA DI AUTENTICAZIONE JWT e viene richiesto il *token JWT* nell'header della richiesta HTTP.

Per prima cosa, vengono presi i valori di *title*, *content* e *category* dal *body* della richiesta.

Successivamente, all'interno di un blocco *try-catch*, viene chiamata la funzione *createNote()*, per creare una nota associata all'utente nel database, anche questa dichiarata nel file [server-utils.js](#).

Se tutto va a buon fine, il server risponde con un messaggio di successo e relativo codice (200), altrimenti, se si verifica un errore generico, esso risponde con un messaggio di errore e relativo codice (500)

- Endpoint GET per ritornare le NOTE ASSOCIATE ALL'UTENTE → [/api/getNotes](#)

Quando viene effettuata una richiesta a questo endpoint, viene utilizzata la STRATEGIA DI AUTENTICAZIONE JWT e viene richiesto il *token JWT* nell'header della richiesta HTTP.

Per prima cosa, viene preso il valore di *category* dai parametri *query* della richiesta, così da ritornare le note coerenti alla categoria, in base alla sezione della dashboard in cui l'utente si trova.

Successivamente, all'interno di un blocco *try-catch*, viene chiamata la funzione *getNotes()*, creata nel file [server-utils.js](#) a cui vengono passati l'*id* dell'utente e la categoria.

Se tutto va a buon fine, il server risponde con un messaggio di successo e relativo codice (200), altrimenti, se si verifica un errore generico, esso risponde con un messaggio di errore e relativo codice (500).

- Endpoint DELETE per ELIMINARE UNA NOTA → [/api/deleteNote/:id](#)

Quando viene effettuata una richiesta a questo endpoint, viene utilizzata la STRATEGIA DI AUTENTICAZIONE JWT e viene richiesto il *token JWT* nell'header della richiesta HTTP.

All'interno di un blocco *try-catch*, viene chiamata la funzione *deleteNote()*, creata nel file [server-utils.js](#) a cui viene passato l'*id* della nota da eliminare.

Se tutto va a buon fine, il server risponde con un messaggio di successo e relativo codice (200), altrimenti, se si verifica un errore generico, esso risponde con un messaggio di errore e relativo codice (500).

- Endpoint PUT per MODIFICARE UNA NOTA → [/api/editNote/:id](#)

Quando viene effettuata una richiesta a questo endpoint, viene utilizzata la STRATEGIA DI AUTENTICAZIONE JWT e viene richiesto il *token JWT* nell'header della richiesta HTTP.

Per prima cosa, vengono presi i valori di *title* e *content* dal *body* della richiesta.

All'interno di un blocco *try-catch*, viene chiamata la funzione *editNote()*, creata nel file [server-utils.js](#) a cui vengono passati *title*, *content* e l'*id* della nota da modificare.

Se tutto va a buon fine, il server risponde con un messaggio di successo e relativo codice (200), altrimenti, se si verifica un errore generico, esso risponde con un messaggio di errore e relativo codice (500).

- Endpoint GET per ritornare l'UTENTE AUTENTICATO → [/api/user](#)

Quando viene effettuata una richiesta a questo endpoint, viene utilizzata la STRATEGIA DI AUTENTICAZIONE JWT e viene richiesto il *token JWT* nell'header della richiesta HTTP.

In questo endpoint, il server risponde semplicemente con un oggetto contenente le informazioni dell'utente autenticato e relativo codice (200).

- Endpoint GET per eseguire la DISCONNESSIONE → [/api/logout](#)

Quando viene effettuata una richiesta a questo endpoint, viene utilizzata la STRATEGIA DI AUTENTICAZIONE JWT e viene richiesto il *token JWT* nell'header della richiesta HTTP.

Viene effettuato il logout dell'utente utilizzando la funzione *req.logout()*.

Se tutto va a buon fine, il server risponde con un messaggio di successo e relativo codice (200), altrimenti, se si verifica un errore generico, esso risponde con un messaggio di errore e relativo codice (500).

- Endpoint GET per la VALIDAZIONE DEL TOKEN JWT → [/api/validateToken](#)

Quando viene effettuata una richiesta a questo endpoint, viene utilizzata la STRATEGIA DI AUTENTICAZIONE JWT e viene richiesto il *token JWT* nell'header della richiesta HTTP.

Il middleware *passport.authenticate('jwt', ...)* verifica se il token JWT presente nella richiesta è valido, quindi l'utente è autenticato.

Se il token è valido, il server risponde con un messaggio di successo e relativo codice (200), altrimenti, se si verifica un errore generico, esso risponde con un messaggio di errore e relativo codice (500).

### *File server-utils.js*

All'interno di questo file, sono state create diverse funzioni, una per ogni query effettuata al database, che vengono esportate ed importate nel file `server.js` dove e quando serve.

Per effettuare le varie query al database, viene utilizzata la funzione `db.query()` fornita dalla libreria `pg` per PostgreSQL.

Le funzioni dichiarate all'interno di questo file sono le seguenti:

- `async function findUser(email)`

Viene eseguita una query SQL asincrona al database per cercare un utente con l'indirizzo email specificato.

```
SELECT * FROM users WHERE email = $1
con array di parametri [email]
```

Il valore `email` viene passato come un array di parametri alla query per evitare possibili attacchi di *SQL injection* (poi inserito al posto di `$1`).

Una volta che la query è stata eseguita con successo, viene restituito il primo risultato trovato dalla query.

- `async function findUserById(id)`

Viene eseguita una query SQL asincrona al database per cercare un utente con l'indirizzo email specificato.

```
SELECT * FROM users WHERE id = $1
con array di parametri [id]
```

Il valore `id` viene passato come un array di parametri alla query per evitare possibili attacchi di *SQL injection* (poi inserito al posto di `$1`).

Una volta che la query è stata eseguita con successo, viene restituito il primo risultato trovato dalla query.

- `async function createUser(email, username, password)`

Viene eseguita una query SQL asincrona per registrare un utente all'interno del database, dati `email`, `username` e `password`.

```
INSERT INTO users (email, username, password) VALUES ($1, $2, $3)
con array di parametri [email, username, password]
```

I valori *email*, *username* e *password* vengono passati come un array di parametri per evitare possibili attacchi di *SQL injection* (poi inseriti al posto di *\$1*, *\$2*, *\$3*)

- *async function* **createNote(title, content, user\_id, category)**

Viene eseguita una query SQL asincrona per creare una nota associata ad un utente all'interno del database, dati *title*, *content*, *user\_id* e *category*.

```
INSERT INTO notes (title, content, user_id, category) VALUES ($1, $2, $3, $4)
con array di parametri [title, content, user_id, category]
```

I valori *title*, *content*, *user\_id* e *category* vengono passati come un array di parametri per evitare possibili attacchi di *SQL injection* (poi inseriti al posto di *\$1*, *\$2*, *\$3*, *\$4*)

- *async function* **getNotes(user\_id, category)**

Viene eseguita una query SQL asincrona per restituire le note associate ad un utente per categoria.

```
SELECT * FROM notes WHERE user_id = $1 AND category = $2
con array di parametri [user_id, category]
```

I valori *user\_id* e *category* vengono passati come un array di parametri per evitare possibili attacchi di *SQL injection* (poi inseriti al posto di *\$1*, *\$2*).

Una volta che la query è stata eseguita con successo, viene restituito il risultato trovato dalla query.

- *async function* **deleteNote(id)**

Viene eseguita una query SQL asincrona per eliminare una nota all'interno del database, dato l'*id* della nota.

```
DELETE FROM notes WHERE id = $1
con array di parametri [id]
```

I valori *title*, *content*, *user\_id* e *category* vengono passati come un array di parametri per evitare possibili attacchi di *SQL injection* (poi inseriti al posto di *\$1*, *\$2*, *\$3*, *\$4*)



Il valore *id* viene passato come un array di parametri alla query per evitare possibili attacchi di *SQL injection* (poi inserito al posto di *\$1*).

- *async function* ***editNote(title, content, id)***

Viene eseguita una query SQL asincrona per modificare una nota all'interno del database, dati *title*, *content* e l'*id* della nota.

```
UPDATE notes SET title = $1, content = $2 WHERE id = $3  
con array di parametri [title, content, id]
```

I valori *title*, *content* e *id* vengono passati come un array di parametri per evitare possibili attacchi di *SQL injection* (poi inseriti al posto di *\$1*, *\$2*, *\$3*).

## Deploy locale con ngrok

### *Problema: Server non raggiungibile*

Il server *Express*, avviato in locale e accessibile all'indirizzo <http://localhost:5000>, non può essere raggiunto da richieste HTTP eseguite da dispositivi esterni, come ad esempio un dispositivo mobile, anche se connessi alla stessa LAN.

Il problema sopra citato, rende inutilizzabile l'applicazione e di conseguenza impedisce di testare/utilizzare l'app su un dispositivo mobile.

### *Soluzione: Utilizzo di ngrok*

La soluzione al problema, applicata a questa applicazione, sta nell'utilizzo di ***ngrok***, ovvero uno strumento utile per creare tunnel sicuri da internet al computer locale.

Nella web app in questione, viene utilizzato *ngrok* per eseguire il tunneling dell'indirizzo <http://localhost:5000>.

Questo ci consente di esporre il nostro server locale *Express* su internet, rendendolo accessibile tramite un URL pubblico temporaneo generato da *ngrok*.

Le richieste HTTP da parte del client, al server, verranno eseguite all'URL generato da *ngrok*.

# DATABASE

## PostgreSQL e pgAdmin

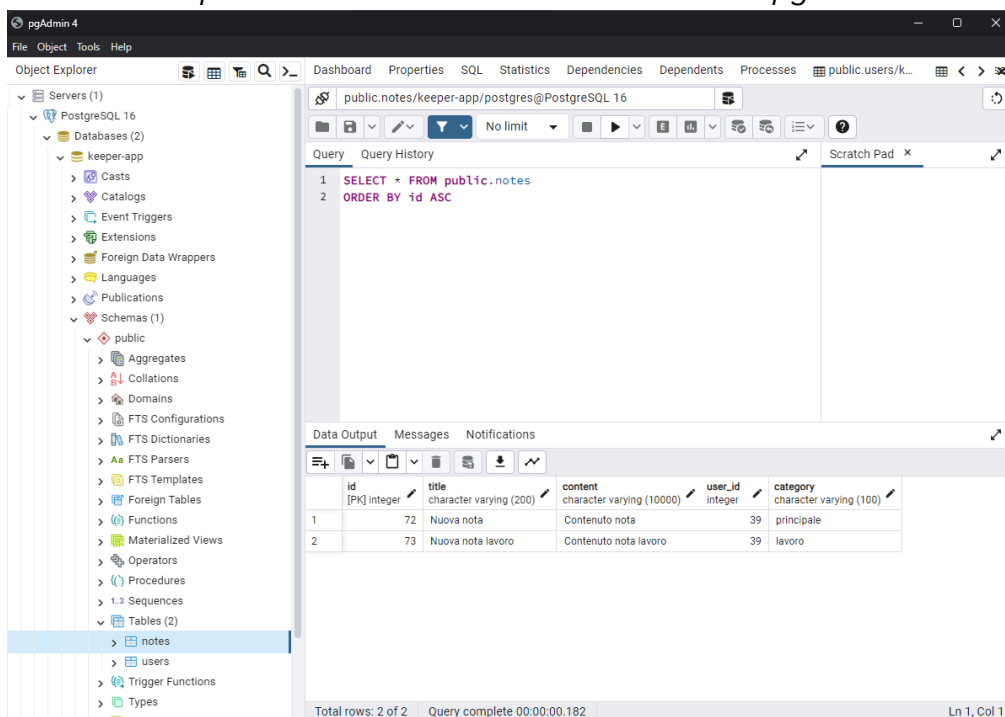
L'applicazione è connessa ad un database relazionale (RDBMS) **PostgreSQL** progettato per fornire alte prestazioni e scalabilità.

Per amministrare e sviluppare il database, viene utilizzato un software chiamato **pgAdmin** progettato per fornire un'interfaccia grafica intuitiva e potente per gestire i database compatibili con PostgreSQL.

Alcune delle caratteristiche principali di *pgAdmin* includono:

- Interfaccia utente intuitiva che permette di esplorare facilmente i database, visualizzare le tabelle ed esaminare i dati all'interno di esse.
- Editor SQL integrato che consente agli utenti di scrivere, modificare ed eseguire query SQL complesse e visualizzare i risultati in un formato tabellare o di altra forma, direttamente dall'interfaccia dell'applicazione
- Strumenti per eseguire il backup e il ripristino dei database PostgreSQL

### *Esempio visualizzazione tabella notes tramite pgAdmin*



The screenshot displays the pgAdmin 4 interface. On the left, the 'Object Explorer' shows the database structure, with the 'notes' table selected under the 'public' schema. The main window shows the 'Query' editor with the following SQL query:

```
1 SELECT * FROM public.notes
2 ORDER BY id ASC
```

Below the query editor, the 'Data Output' pane displays the results of the query in a table format:

id	title	content	user_id	category
72	Nuova nota	Contenuto nota	39	principale
73	Nuova nota lavoro	Contenuto nota lavoro	39	lavoro

The status bar at the bottom indicates 'Total rows: 2 of 2' and 'Query complete 00:00:00.182'.

## Tabelle

Il database *keeper-app* è formato dalle seguenti **TABELLE**:

- **notes** → contiene le note scritte dai vari utenti
- **users** → contiene gli utenti registrati

La tabella users contiene le seguenti colonne:

- id → *primary key* - numero univoco auto-incrementale che identifica l'utente
- email → email inserita dall'utente in fase di registrazione
- username → username inserito dall'utente in fase di registrazione
- password → hash della password inserita dall'utente in fase di registrazione

La tabella notes contiene le seguenti colonne:

- id → *primary key* - numero univoco auto-incrementale che identifica la nota creata
- title → titolo della nota
- content → contenuto della nota
- user\_id → *foreign key* - identifica l'utente che ha scritto la nota
- category → categoria a cui fa parte la nota

Le due tabelle sono state create scrivendo il seguente codice SQL nell'editor di pgAdmin:

<pre>CREATE TABLE users (     id SERIAL PRIMARY KEY,     email VARCHAR(100),     username VARCHAR(100),     password VARCHAR(100) );</pre>	<pre>CREATE TABLE notes (     id SERIAL PRIMARY KEY,     title VARCHAR(200),     content VARCHAR(10000),     user_id INTEGER,     category VARCHAR(100),     FOREIGN KEY (user_id) REFERENCES         users(id) );</pre>
--	--

La colonna *id* per entrambe le tabelle è definita come SERIAL, che crea automaticamente una sequenza e imposta la colonna come auto-incrementale e univoca.

## Connessione al database

La **CONNESSIONE** al database nell'applicazione viene gestita tramite una libreria comunemente utilizzata per interagire con un database PostgreSQL chiamata *pg*.

Essa fornisce alcune funzioni per eseguire query SQL e altre operazioni di gestione del database.

### Creazione istanza db

Viene creata un'istanza del client PostgreSQL chiamata *db*, che viene utilizzata per eseguire query SQL e interagire con il database *keeper-app* sul server locale.

```
const db = new pg.Client({
  user: 'postgres',
  host: 'localhost',
  database: 'keeper-app',
  port: 5432,
  password: process.env.DB_PASSWORD
});
```

Il client viene configurato con i dettagli di connessione al database:

- *user* → specifica l'utente del database, di default è "postgres"
- *host* → specifica l'host del database, che è "localhost", essendo hostato in locale
- *database* → specifica il nome del database, che in questo caso è "keeper-app"
- *port* → specifica la porta su cui il database è in ascolto, di default è 5432
- *password* → specifica la password per l'utente, si trova nel file *.env*

### Connessione

La funzione *db.connect()* viene utilizzata per avviare la connessione al database, e una volta che la connessione è stata stabilita oppure si verifica un errore durante il tentativo di connessione, viene eseguita la *funzione di callback* fornita.

```
db.connect((err) => {
  if (err) {
    console.error('Errore di connessione', err.stack)
  } else {
    console.log('Connesso al database Keeper-App')
  }
});
```

Se si verifica un errore durante il tentativo di connessione, viene stampato un messaggio di errore sulla console, che include lo stack di errore per facilitare la diagnosi del problema.

Altrimenti, viene stampato un messaggio di conferma sulla console, indicando che la connessione al database è stata stabilita con successo.

## Esecuzione query

Per eseguire una *query SQL* al database, viene utilizzato il metodo *query()* dell'oggetto *db*, all'interno di un blocco *try-catch* per gestire eventuali errori.

### Esempio di query SQL

```
try {
  const results = await db.query('SELECT * FROM users WHERE email = $1', [email]);
  return results.rows[0];
} catch (error) {
  console.error('Errore nella query findUser:', error);
}
```

Nell'applicazione, le query vengono eseguite all'interno di funzioni apposite create nel file [server-utils.js](#) e poi importate nel file *server.js* quando e dove necessario.

## Hashing delle password

Per **HASHING** delle password si intende una pratica sicura per proteggere le password degli utenti, in quanto rende molto difficile per i malintenzionati decifrare le password, anche se riescono ad accedere al database.

Per creare una password hashata partendo da una password plain-text, viene utilizzata la libreria *bcrypt*.

### Creazione hash

La creazione dell'hash partendo dalla password plain-text, viene eseguita in fase di registrazione di un utente.

Per fare ciò, viene utilizzata una funzione apposita, fornita da *bcrypt*, chiamata *hash()*, che accetta due parametri:

- *password* → la password plain-text inserita dall'utente che si desidera hashare
- *salting rounds* → numero di iterazioni dell'algoritmo di hashing

Nel caso dell'applicazione in questione, il numero di *salting rounds* impostato è 10; un costo più elevato renderebbe l'hashing più sicuro, ma viene richiesto più tempo per completare l'operazione.

---

Una volta creato l'hash, viene salvato nel database, assieme al resto delle informazioni fornite.

```
const hash = await bcrypt.hash(password, 10);  
await createUser(email, username, hash);
```

### Comparazione password

Per eseguire la comparazione delle password viene utilizzata la funzione *compare()*, fornita da *bcrypt* che confronta la password inserita dall'utente con l'hash memorizzato nel database utilizzando un algoritmo di confronto.

La funzione accetta due parametri e restituisce *true* se le due password corrispondono, altrimenti restituisce *false*:

- *password* → password inserita dall'utente in fase di accesso
- *user.password* → hash della password memorizzato nel database per l'utente.

```
const isValidPassword = await bcrypt.compare(password, user.password);
```

## TESTING e VERSIONING

### Test sviluppatore

In questa fase di test, è stata utilizzata l'applicazione direttamente dallo sviluppatore, in modo da verificare la correttezza, la coerenza e il corretto funzionamento del codice.

Le verifiche eseguite sono state:

- Verifica fase di **registrazione**:
  - Test registrazione di un nuovo account ✓
  - Test di inserimento dati non validi in fase di registrazione, come ad esempio l'inserimento di una password diversa dalla password inserita nel campo di conferma password ✓
  - Test registrazione di un utente già registrato ✓
- Verifica fase di **accesso**:
  - Test accesso ad un account registrato ✓
  - Test di inserimento dati non validi in fase di accesso, come ad esempio l'inserimento di una password errata ✓
  - Test accesso ad un account non esistente nel database ✓
- Verifiche fase di **aggiornamento/visualizzazione/eliminazione dati**:
  - Test creazione/modifica/eliminazione di una nota ✓
  - Test visualizzazione note separate in base alla categoria ✓
  - Test validità *Token JWT* e accesso a zone dell'applicazione senza previa autenticazione ✓

### Test endpoints con Postman

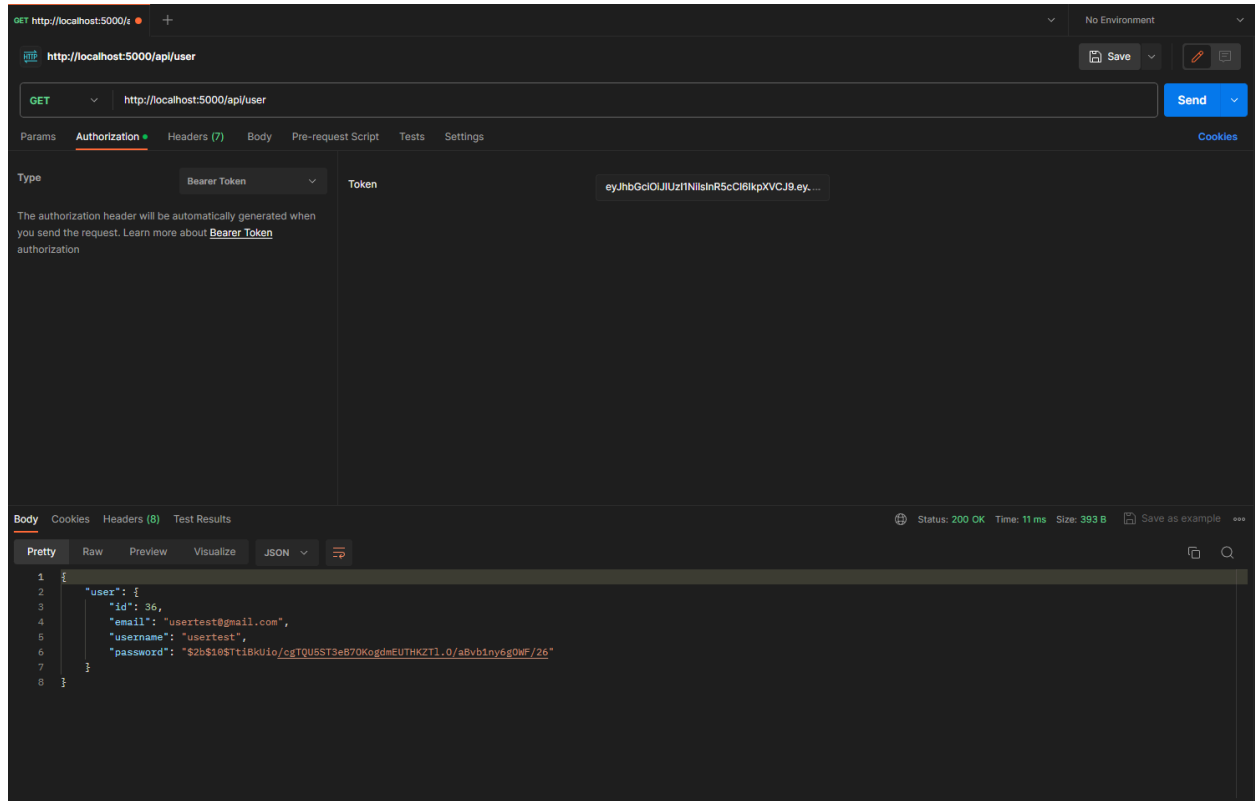
Per verificare che gli endpoints nel server *Express.js* svolgano correttamente le azioni per cui sono stati specificati, è stato utilizzato un software esterno chiamato **POSTMAN**.

Postman è un software che facilita lo sviluppo, il test e la documentazione delle API (Application Programming Interface).

Esso consente di creare facilmente diverse tipologie di richieste HTTP, come GET, POST, PUT, DELETE e altre, utilizzando un'interfaccia grafica intuitiva.

Inoltre, è possibile specificare gli header, i parametri, il corpo della richiesta e altri dettagli in modo semplice e chiaro.

*Esempio test endpoint `/api/user` con Postman e relativo risultato*



## Test utente

L'applicazione è stata testata da persone inesperte, che non conoscono la logica e le procedure da seguire per utilizzarla.


Agli **UTENTI GENERICI** che hanno eseguito il test non è stato dato alcun suggerimento particolare, è stato solo chiesto di utilizzare l'applicazione ed effettuare:

- Registrazione del proprio account nel database
- Accesso al proprio account
- Creazione di una nota (per ogni categoria) ed eventuale modifica/eliminazione di essa
- Disconnessione dall'account

Non ci sono stati particolari problemi nell'utilizzo dell'applicazione, lo scopo principale di questo test è stato quello di ascoltare commenti da parte degli utilizzatori, in modo da apportare potenziali modifiche all'applicazione, se necessarie.

## GitHub





Per la gestione del codice sorgente e il controllo delle versioni, è stato utilizzato GitHub, dove è stato possibile tenere traccia delle modifiche apportate al codice nel tempo, committando e pushando regolarmente al repository.

[\*Keeper App - Github\*](#)

Nel codice sorgente è presente un file chiamato *.gitignore*, ovvero un file di configurazione utilizzato nel sistema di versioning Git per specificare quali file e cartelle devono essere ignorati e non devono essere inclusi nel repository Git.

---

## POSSIBILI ESTENSIONI FUTURE

### OAuth con Google

L'applicazione in un futuro potrebbe supportare un sistema di autenticazione **OAuth (Open Authorization)** come quello di Google, oltre al sistema email/password già presente.

L'implementazione è possibile grazie all'utilizzo di *Passport.js* che fornisce il supporto per l'autenticazione OAuth con Google tramite il modulo *passport-google-oauth* o *passport-google-oauth20*.

Dopo aver dato un'occhiata alla documentazione fornita da *Passport.js*, un'ipotetica implementazione potrebbe essere la seguente:

1. Registrazione dell'applicazione su Google Developers Console per ottenere le credenziali OAuth (ID e chiave segreta) necessarie per configurare l'autenticazione
2. Configurazione di *Passport.js*, (basandosi sulla documentazione) per utilizzare l'autenticazione OAuth con Google
3. Configurazione endpoints nel server per gestire le richieste di autenticazione con Google (basandosi sulla documentazione)

### Più categorie di note

L'applicazione in un futuro potrebbe permettere all'utente di scrivere note in diverse altre categorie, oltre alle due già presenti.

Oppure, ancora meglio, potrebbe permettere all'utente di creare categorie personalizzate e scrivere note all'interno di esse.

### Switch modalità chiara-scura

L'applicazione attualmente ha un design scuro, ma in un futuro potrebbe essere inserito un pulsante che funge da switch per permettere all'utente di intercambiare tra modalità chiara e scura.

Una delle possibili implementazioni di questa funzionalità potrebbe essere la seguente:

1. Nei vari fogli di stile associati ai componenti, si vanno a creare delle classi CSS aggiuntive che applichino un design chiaro ai vari elementi dell'interfaccia

- 
2. Ipotizzando di creare il pulsante nel componente *Home.jsx*, si procede nel seguente modo:
    - a. Si crea uno stato booleano (ad es. *isLightModeOn*), inizialmente *false*
    - b. Si crea una funzione (ad es. *handleModeClick*), assegnata all'evento *onClick* del pulsante che, ad ogni click, imposta il valore della variabile *isLightModeOn* all'opposto del suo valore attuale (switch)
  3. In base al valore della variabile *isLightModeOn*, i vari componenti vengono renderizzati o con lo stile CSS della modalità scura o con lo stile CSS della modalità chiara, un esempio potrebbe essere:

```
<div className={isLightModeOn ? "note-light" : "note"}>...</div>
```