

Don't
waste
your
time!

Don't
give
up!

BE
yourself

NOTESHARING1.0

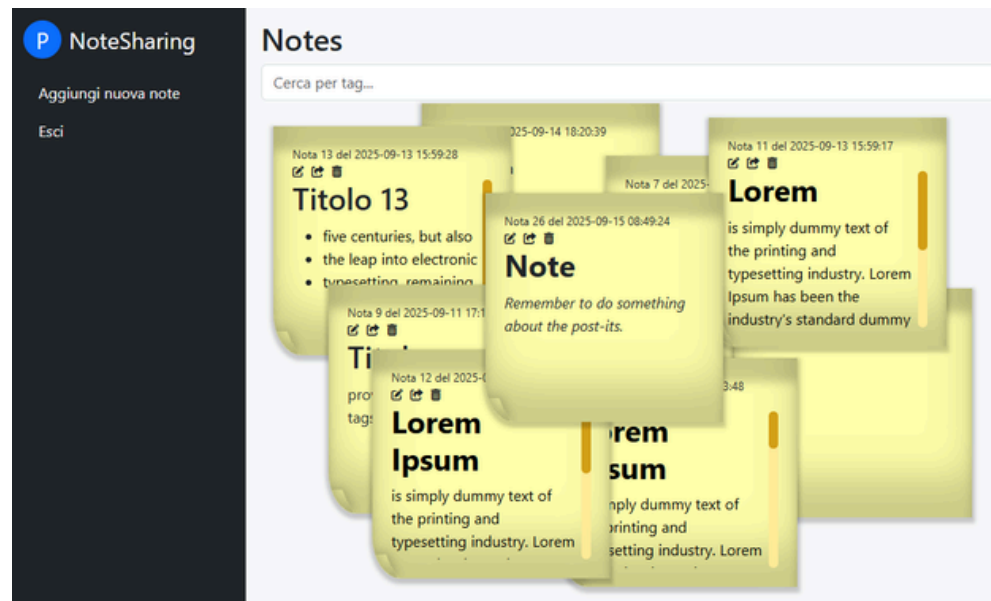
Powered by:

Giuseppe De Martino

Ph. D. (Thesis defence)

Abstract

Il presente documento ha lo scopo di illustrare il progetto richiesto per la selezione del Dipartimento per la Transizione Digitale. L'obiettivo è la realizzazione di una web app per la condivisione di note, corredato dalle soluzioni implementative e dalle strategie adottate.



Premesse

Data la natura del progetto e il tempo limitato a disposizione, unito al totale impegno profuso nei confronti della mia azienda, sono state implementate tutte le funzionalità richieste, ad eccezione dell'integrazione con MongoDB.

L'architettura backend è stata ideata da me e è stata pensata per configurare ed esporre servizi REST.

Questa architettura prevede l'implementazione di diversi pattern, tra cui il reflection delle classi, e risulta già in uso in diversi applicativi a me commissionati durante i miei ultimi anni di attività lavorativa. Grazie alla sua flessibilità, può essere adattata anche a contesti B2B.

Inoltre, in questo documento verranno presentati anche alcuni possibili sviluppi futuri, con particolare riferimento a come MongoDB sarebbe stato impiegato e integrato.

Infine, si precisa che il testo e la revisione grammaticale del presente documento sono stati effettuati con il supporto di ChatGPT.

L'intero progetto è scaricabile su <https://github.com/demnap-work/notesharing>

Indice

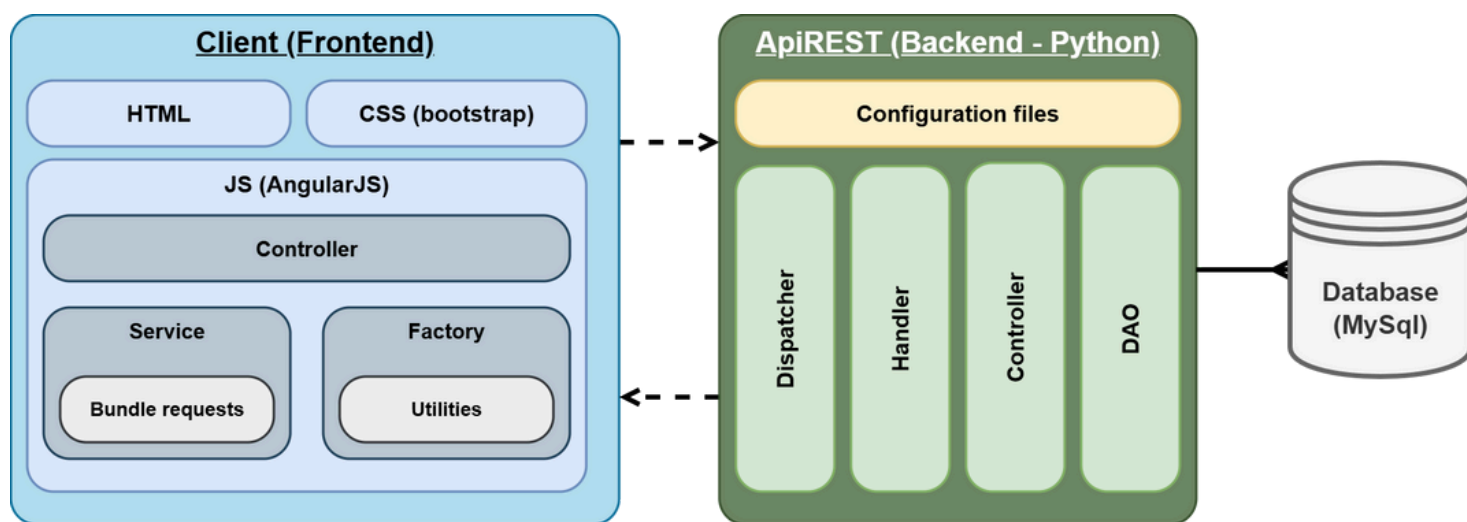
- Descrizione del progetto
- Servizi APIRest HTTP
- Descrizione dell'architettura frontend
- Descrizione dell'architettura backend
- Gestione delle sessione - Token
- Configurazione e avvio
- Schema database

Descrizione del progetto

Il progetto consiste in una Single Page Application (SPA) che richiama servizi API REST esposti da un backend locale sviluppato in Python.

Il frontend è stato realizzato utilizzando HTML e CSS, con il supporto di AngularJS per la gestione delle logiche applicative. Per la parte grafica è stata adottata la libreria Bootstrap 5, che ha consentito di strutturare un'interfaccia responsive e uniforme.

Di seguito viene riportata l'architettura ad alto livello del frontend e del backend:



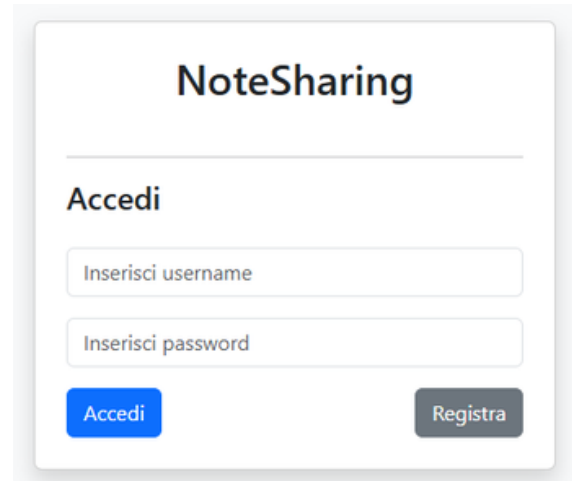
Per questo progetto, sul lato frontend, sono state sviluppate diverse funzionalità che permettono la gestione completa delle note e l'interazione tra utenti. In particolare:

- Login con username e password (come richiesto)
- Registrazione di nuovi utenti
- Visualizzazione delle note (create e condivise)
- Inserimento nuova nota
- Filtraggio delle note tramite tag
- Modifica nota
- Eliminazione nota
- Condivisione nota con altri utenti

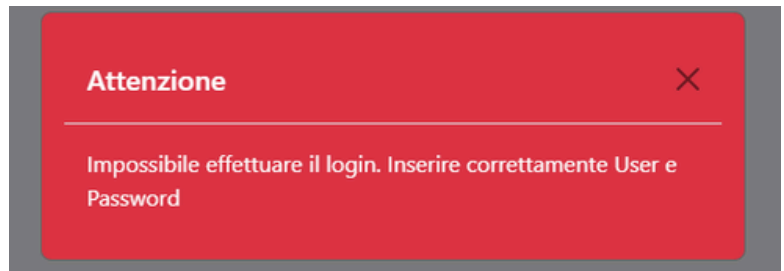
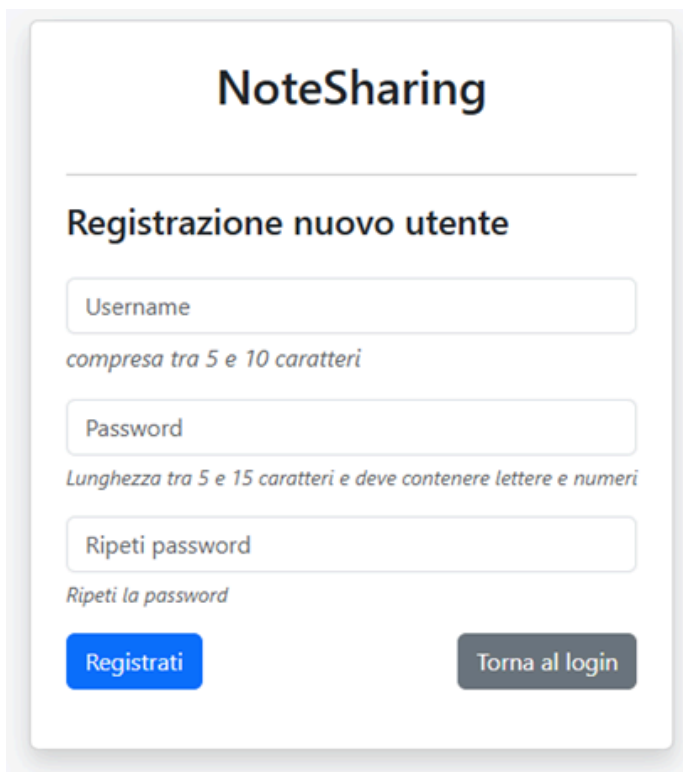
Login

La schermata di login prevede l'inserimento di uno username (non necessariamente un indirizzo email), la cui lunghezza deve essere compresa tra un minimo di 5 e un massimo di 15 caratteri.

La password deve invece avere una lunghezza compresa tra 5 e 10 caratteri e includere sia lettere che numeri. Una volta effettuato il login, compare la schermata di home, che in questo caso assume le sembianze di una dashboard.



In caso di password errata, comparirà il seguente messaggio di errore. In generale, i messaggi di errore e qualsiasi altra comunicazione dell'applicazione verso l'utente avranno la seguente forma:

Registrazione di nuovi utenti

Dalla schermata di login è inoltre possibile accedere alla schermata di registrazione.

Figura 1 Schermata di login

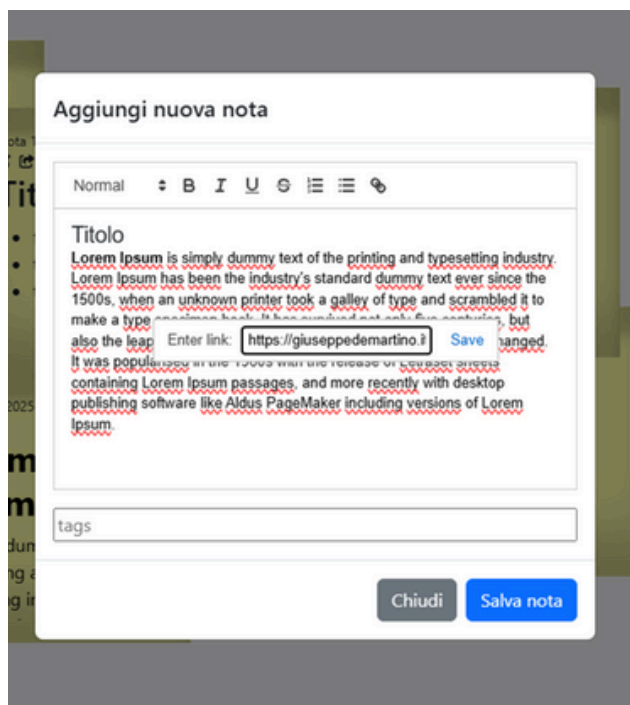
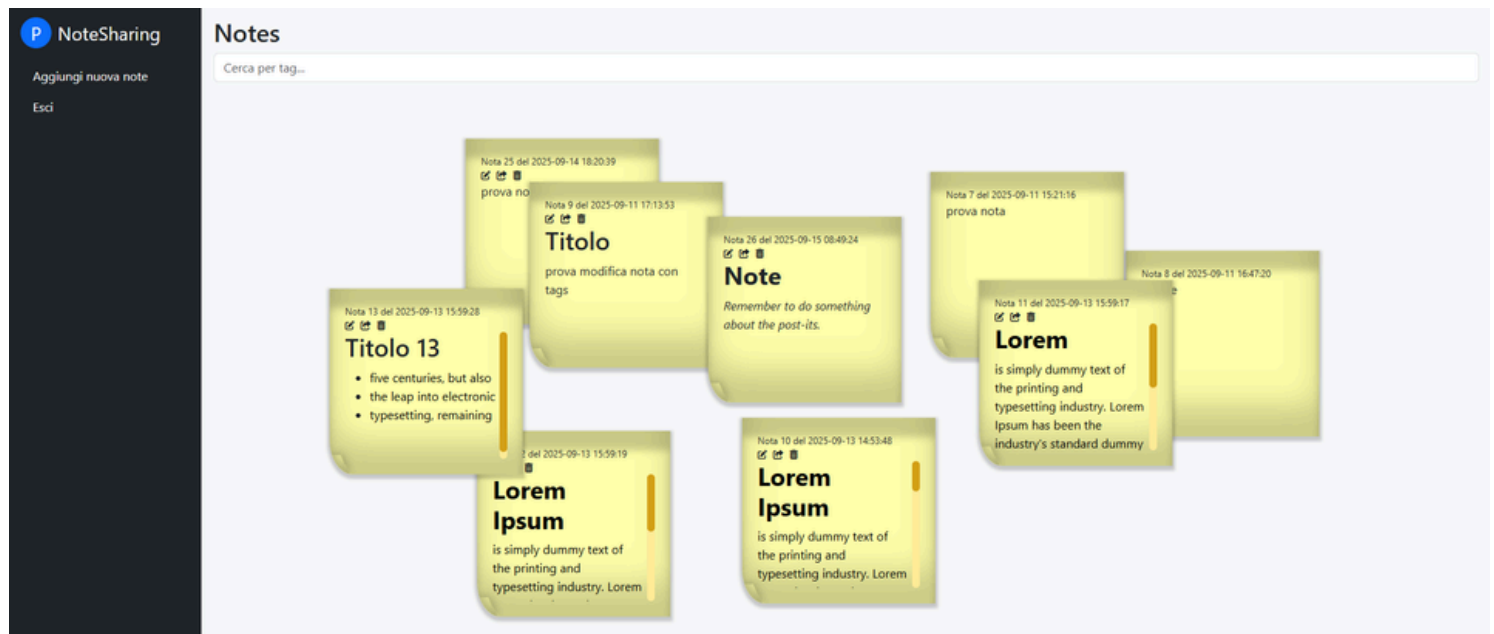
Come si può notare, per questo progetto si è scelto di utilizzare il minor numero di campi richiesti in fase di autenticazione, al fine di concentrare il maggiore effort sulle altre funzionalità, in particolare sulla gestione delle note.

Visualizzazione delle note

Se il login va a buon fine si accede alla schermata principale in cui le note sono rappresentate come fogli di post-it, con la possibilità di essere spostate liberamente all'interno del container.

Sul lato sinistro della schermata è presente una sidebar che raccoglie l'elenco delle funzionalità: *Aggiungi nuova nota* e *Esci* (logout).

In alto al centro è presente anche la casella di testo per filtrare le note per tags.



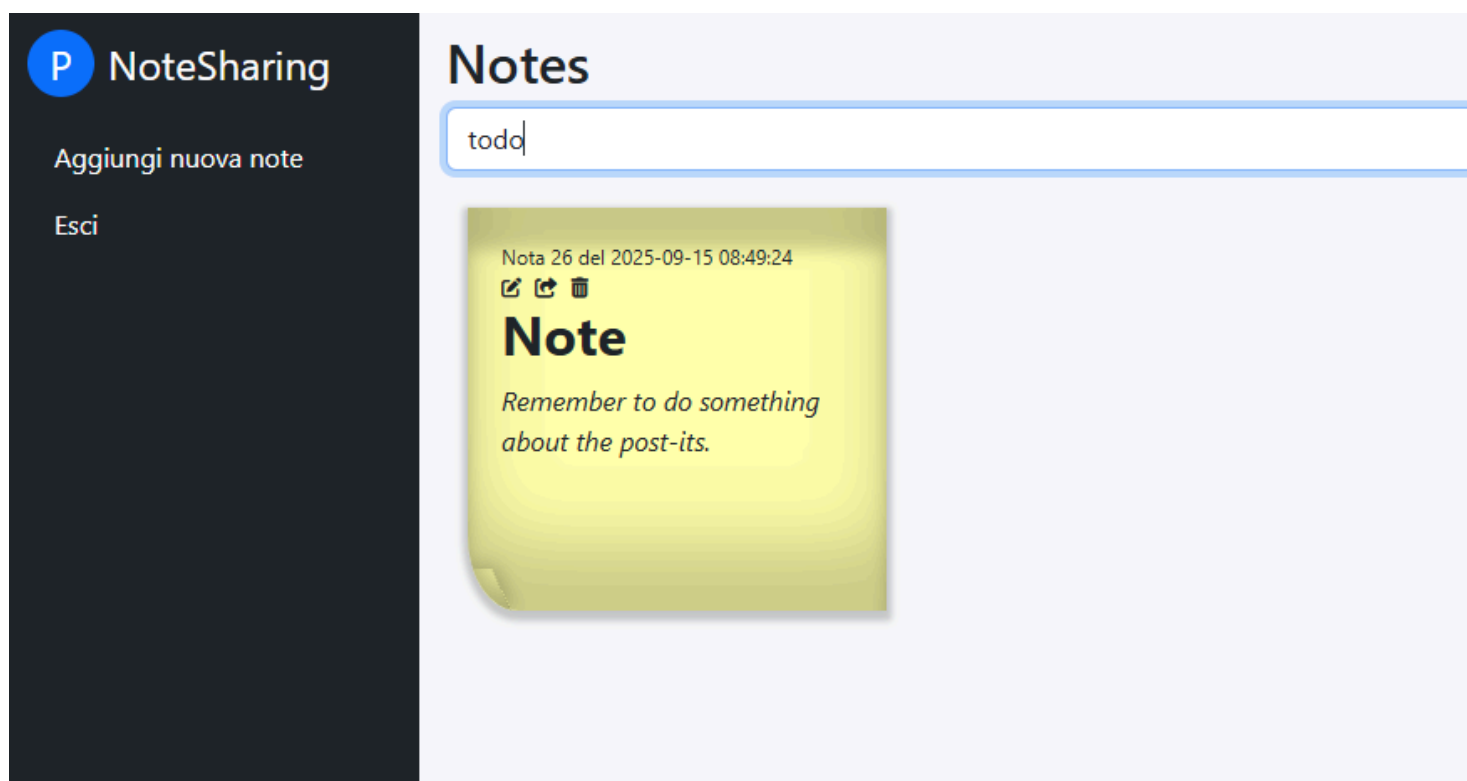
Inserimento nuova nota

L'inserimento di una nuova nota è gestito graficamente attraverso un editor di testo realizzato con un tool JavaScript open source e integrato all'interno di un modale, come mostrato in figura.

L'editor offre le seguenti funzionalità per la formattazione del testo come e di esportarlo in formato html fra cui anche la possibilità di inserire collegamenti a fonti esterne.

Filtraggio delle note tramite tag

Il filtraggio delle note avviene tramite casella di ricerca posta in alto ed è gestita esclusivamente lato frontend: durante la digitazione vengono immediatamente selezionate tutte le note contenenti i caratteri inseriti nella casella di ricerca, mentre le altre scompaiono temporaneamente dalla schermata. Per riportare tutte le note in visualizzazione, è sufficiente cancellare il contenuto della casella di ricerca.



Funzionalità nota



La nota può essere modificata, condivisa ed eliminata soltanto se si è il creatore, in questo caso sulla nota saranno visualizzate 3 icone relative alla modifica, condivisione e eliminazione della stessa.

Modifica nota



Al click sull'icona modifica presente sulla nota, verrà visualizzata la stessa schermata di inserimento nuova nota con lo stesso editor popolato con i dettagli della nota e dei tags.

Elimina nota



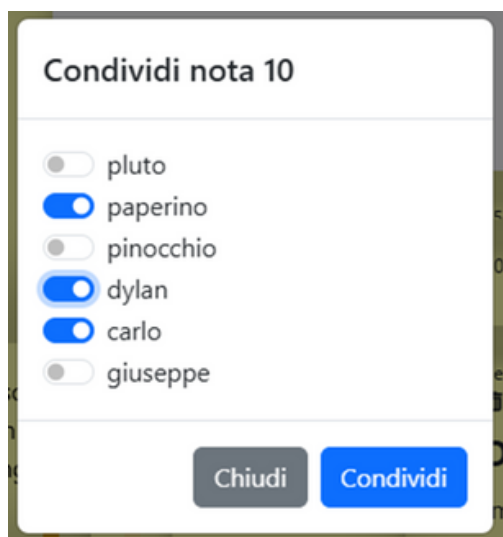
Al click sull'icona del cestino è possibile eliminare la nota, comparirà una schermata di conferma eliminazione come mostrato in figura. Una volta eliminata la nota, quest'ultima non comparirà tra le note condivise con altri utenti



Condividi nota



Al click sull'icona condivisione, sempre presente sulla nota, apparirà la schermata con tutti gli utenti a cui si desidera condividere la nota come mostrato in figura a sinistra.



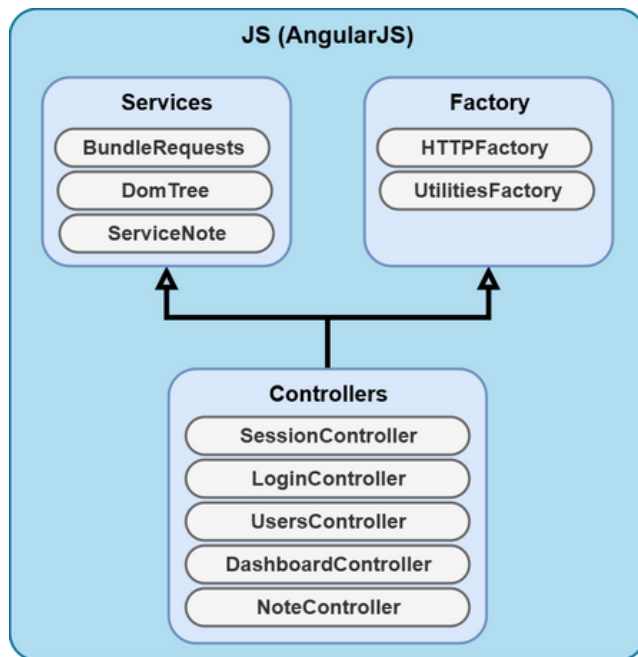
Servizi API Rest HTTP

Di seguito vengono riportati gli end point ai servizi API

Path	Metodo	Header richiesti	Parametri Body	Tipo	Descrizione
/api/checkTokenAlive	GET	token (string, required)	–	–	Verifica se il token di sessione è ancora valido.
/api/login	POST	–	user (string, required) pwd (string, required)	form-data	Effettua login e restituisce un token di autenticazione.
/api/logout	GET	token (string, required)	–	–	Effettua il logout e invalida il token di sessione.
/api/addUser	POST	–	user (string, required) pwd (string, required) pwd2 (string, required)	form-data	Registra un nuovo utente con username e password.
/api/getAllUsers	POST	token (string, required)	–	–	Restituisce la lista di tutti gli utenti registrati.
/api/insertNota	POST	token (string, required)	nota (string, required) tags (string, optional)	form-data	Inserisce una nuova nota con eventuali tag.
/api/sharedNota	POST	token (string, required)	idNota (string, required) idUser (string, required)	form-data	Condivide una nota con uno o più utenti specificati.
/api/getAllNotes	GET	token (string, required)	–	–	Recupera tutte le note proprie e quelle condivise con l'utente autenticato.
/api/updateNote	POST	token (string, required)	notaNuova (string, required) idNota (int, required) tags (string, optional)	form-data	Aggiorna una nota esistente modificandone contenuto e tag.
/api/removeNote	POST	token (string, required)	idNota (int, required)	form-data	Elimina una nota esistente dall'archivio.

Descrizione dell'architettura frontend

L'architettura frontend è realizzata con AngularJS v1.4.8, utilizzato per implementare il pattern Model-View. Si tratta di una versione ormai datata, ma che consente di evitare l'installazione di ulteriori servizi richiesti dalle versioni più recenti di Angular. Nell'immagine è riportata l'architettura suddivisa in controller, services e factories.

**HTTPFactory**

Questa classe factory fornisce il wrapper per le chiamate asincrone in ajax tramite il pattern promissse

UtilitiesFactory

Fornisce tutte le funzionalità generiche

DomTree (Service)

Questa classe è l'unica classe delegata ad accedere al dom, fornisce gli oggetti jQuery dei vari elementi della pagina web

bundleRequests (Service)

In questa classe vengono centralizzati tutti gli endpoint alle chiamate API

ServiceNote (Service)

Questa classe fornisce alcune funzionalità relative alle note e che sono condivise con altri controller come la *creazione delle note*

SessionController

Ha il compito di gestire lo stato della sessione utente e l'accesso all'applicazione, la funzione principale è **checkIsTokenAlive(options)**,

- Esegue una chiamata HTTP (GET) all'endpoint `bundleRequest.checkTokenAlive`.
- Passa il token come header della richiesta.
- Se la risposta è positiva (`status == "OK"`):
 - Mostra la dashboard principale e nasconde il form di login.
 - Carica tutte le note tramite `serviceNote.getAllNote()`.
 - Aggiorna l'icona utente (`icoUser`) con l'iniziale dello username.
- Se la risposta è negativa:
 - Mostra il form di login.
- In caso di errore AJAX:
 - Mostra un messaggio di errore generico con `utilitiesFactory.alertMessage()`.

LoginController

Gestisce l'autenticazione degli utenti, mostrando il form di login, inviando le credenziali al backend e aggiornando la UI in base all'esito della richiesta. Inoltre, permette di passare dalla schermata di login al form di registrazione, le funzioni sono:

1.initLogin()

- Logga un messaggio di inizializzazione.
- Probabilmente viene chiamata quando la pagina di login è caricata.

2.login(model)

- Recupera username e password dai campi input `#user` e `#pwd`.
- Invoca la funzione privata `login()` passando le credenziali.

3.showRegistrazione(model)

- Nasconde il form di login.
- Mostra il form di registrazione utente.

UserController

Gestisce la registrazione di nuovi utenti e il passaggio dalla schermata di registrazione a quella di login, le funzioni principali sono:

1. `showLogin()`
 - a. Nasconde il form di registrazione.
 - b. Mostra il form di login.
 - c. Logga "sono qui" in console (probabilmente usato per debug).
2. `addNewUser()`
 - a. Recupera i valori dei campi di input:
 - i. `usernameText`
 - ii. `passwordText`
 - iii. `password2Text`
 - b. Invoca la funzione privata `addNewUser()` con i dati raccolti.

DashboardController

Gestisce le principali funzionalità della dashboard dell'applicazione: logout, ricerca e filtro delle note, apertura dell'editor di nuove note e interazione con i modali di gestione, le funzioni principali sono:

1. `logout(model)`
 - a. Richiama la funzione privata `logout()` passando il token salvato in `sessionStorage`.
2. `initNote(model)`
 - a. Definita ma vuota (probabile punto di estensione per inizializzare le note al caricamento).
3. `searchTag(model)`
 - a. Invoca la funzione privata `searchTag()`, che filtra le note nella dashboard in base ai tag inseriti dall'utente.
4. `deleteNotaShow(model)`
 - a. Mostra il modale di conferma per l'eliminazione di una nota.
5. `openEditorNota(model)`
 - a. Apre il modale dell'editor di note.
 - b. Reset dei campi (editor vuoto, titolo, tag).
 - c. Mostra il pulsante Salva nota e nasconde Modifica nota.

NoteController

Gestisce tutte le operazioni legate alla gestione delle note: creazione, modifica, eliminazione, condivisione e inizializzazione dell'editor di testo.

Funziona come punto di collegamento tra l'interfaccia utente (DOM), l'editor Quill e il backend tramite chiamate API, le funzioni principali sono:

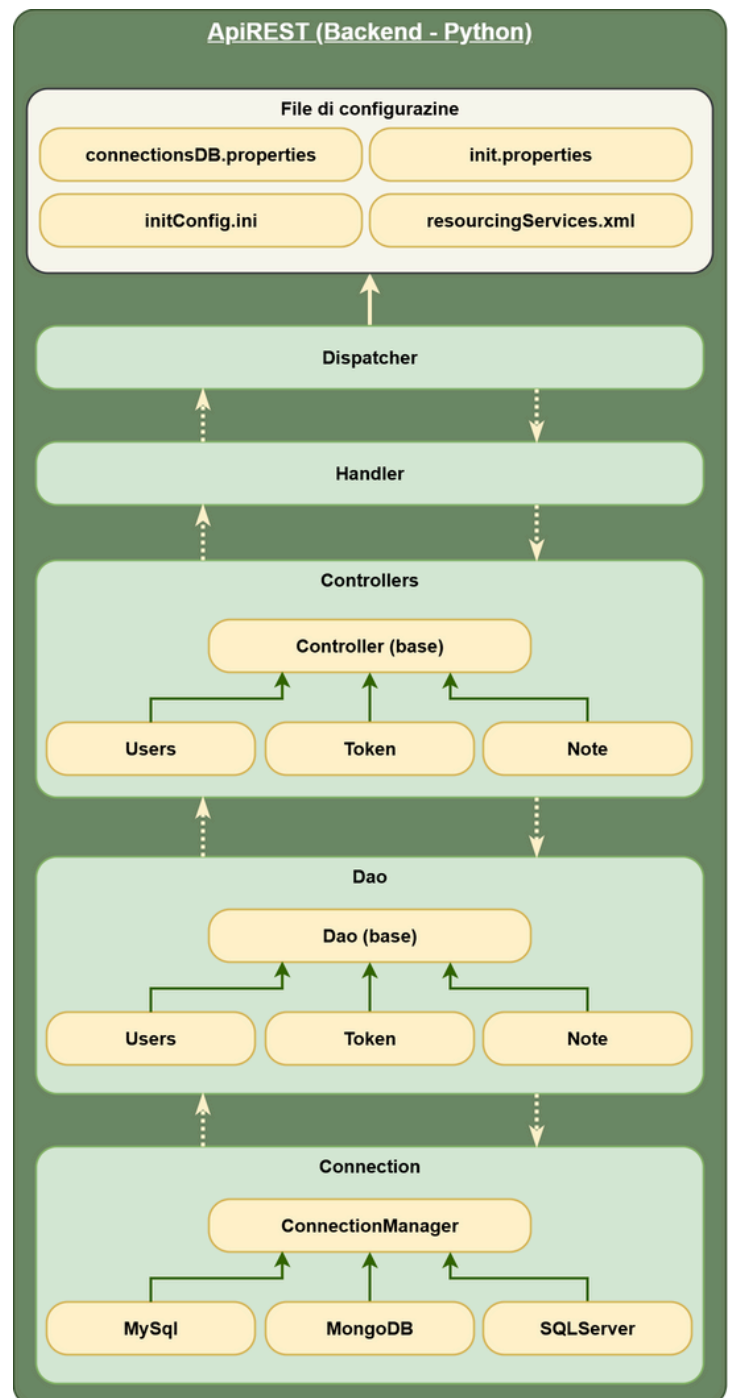
1. `initEditor()`
 - a. Inizializza l'editor Quill con tema Snow e toolbar.
2. `exportHTML()`
 - a. Restituisce l'HTML della nota scritta nell'editor.
3. `salvaNota()`
 - a. Crea una nuova nota.
 - b. Richiama `salvaNota()` passando tags, contenuto della nota (HTML) e token.
4. `updateNote()`
 - a. Aggiorna una nota esistente.
 - b. Richiama `updateNote()` con `idNota`, nuovo contenuto, tags e token.
5. `deleteNota()`
 - a. Elimina una nota esistente.
 - b. Richiama `deleteNota()` passando `idNota` e token.
6. `shareNota()`
 - a. Condivide una nota selezionata con più utenti.
 - b. Recupera la lista degli utenti selezionati con checkbox e richiama `shareNota()`.

Descrizione dell'architettura backend

Il backend, come descritto in precedenza, fornisce servizi API REST ed è implementato in Python. I servizi sono creati e gestiti tramite la libreria Flask. Nell'immagine è rappresentata l'architettura ad alto livello del backend, che include file di configurazione per il comportamento delle API e le connessioni al database.

L'architettura si compone di:

- Dispatcher: prepara l'applicazione utilizzando i dati di configurazione;
- Handler: coordina i vari controller;
- Controller: gestiscono la logica di ciascuna chiamata API;
- DAO (Data Access Object): si occupano dell'accesso e della gestione dei dati nel database.
- Connection Manager: fornisce l'oggetto di connessione al database in base ai dati di configurazione.



Dispatcher.py

Questa classe dispatcher.py è un front controller per tutti i controller ed è sviluppato in Flask: gestisce la configurazione iniziale, espone i servizi REST e centralizza il logging e la gestione degli errori. Essa prevede: Configurazione iniziale (singleton), Usa il filePropertiesSingleton("configFiles/init.properties") per leggere i parametri dal file di configurazione; dal file prende:

- metodi HTTP abilitati (GET, POST ecc.)
- file di log
- nome app, versione, URL base
- host e porta del server
- parametri per la gestione dei token di sessione
- path verso le risorse (resourcingServices).

Poi viene applicato un pattern Singleton: __instance che tiene l'unica istanza della classe in modo tale che il costruttore (__init__) inializza solo la prima volta.

La creazione dell'app Flask avviene tramite l'istanza **Flask(__name__)** e abilita il CORS (**CORS(app, support_credentials=True)**), per le sole esigenze di questo progetto.

L'handler principale espone la route dinamica (unico punto di accesso), così formata:

@app.route("/<p>/<s>", methods=methodsEnabled)

Questo significa che intercetta tutte le chiamate a / p / s , indipendentemente dal servizio, purché il metodo sia abilitato. Inoltre, crea un oggetto di Handler (altra classe vero orchestratore delle business logic), all'interno viene invocata la Chiamata a runServices() della classe Handler.py passando:

- path richiesto
- metodo (GET/POST)
- parametri normalizzati
- token di sessione (ItToken)
- headers.

E restituisce la risposta in JSON (grazie al decoratore @jsonDec.toJson).

Gli errori sono intercettati vari errori HTTP:

- 400 → parametri non validi
- 404 → risorsa non trovata
- 500 → errore imprevisto
- 405 → metodo non consentito

In tutti i casi viene restituito un JSON uniforme:

`{ "status": "KO", "result": "messaggio di errore" }`

Il L'avvio del server avviene mediante il metodo `start()` decide se:

- avviare l'app in HTTP (`sslContext=False`)
- avviare in HTTPS ad-hoc (`sslContext=True`).

Questa classe è il cuore del backend, legge la configurazione, inizializza il server Flask, gestisce in modo centralizzato tutte le richieste REST (routing generico), passa la palla al Handler che implementa la logica, restituisce sempre JSON e garantisce logging e gestione uniforme degli errori.

Handler.py

È il motore centrale di dispatching, prende in carico una richiesta (preparata dal dispatcher) ed effettua i seguenti controlli e azioni:

- risorsa esistente (valida),
- controllo sui parametri obbligatori,
- controllo sul tipo dei parametri,
- verifica che la sessione sia ancora attiva (check su token),
- istanzia il controller associato alla risorsa e lo estende con il controller base,
- riceve la risposta dal controller e formatta la risposta.

Il metodo `runServices(cls, requestAttribute)` è il metodo principale che si occupa di eseguire concretamente una richiesta http e ha il seguente comportamento: prendi in ingresso i seguenti parametri: `path`, `method`, `param`, `header`, `ltToken` (last time del token) e restituisce un JSON, di seguito viene riportato l'algoritmo di una chiamata gestita dal handler:

1. Validare la richiesta con `validRequest`.
 - a. Se ci sono errori di messaggio (`msgErr`), restituire risposta KO.
 - b. Se ci sono errori nei parametri (`paramErr`), restituire risposta KO.
2. Verificare l'esistenza di un `controllerClass`.
3. Caricare e inizializzare il controller base.
4. Se è presente un token nell'`header`:
 - a. Verificare il token con `checkToken`.
 - b. Se valido, controllare scadenza rispetto a `ltToken`.
 - c. Se non scaduto, aggiornare il token e salvare `userInfo`.
 - d. Se non valido o scaduto, restituire KO (token scaduto).
5. Importa dinamicamente il controller specifico e recuperare il metodo da eseguire.
6. Invocare il metodo con gli argomenti e header validati.
 - a. Se si verifica un'eccezione durante l'esecuzione, restituire errore imprevisto.
 - b. Se la risposta del metodo è `None`, restituire errore generico di servizio.
 - c. Altrimenti, restituire la risposta ottenuta dal controller.

Controller

Le classi controller estendono la classe base (interfaccia/astratta) progettata per standardizzare e semplificare la gestione delle chiamate API nei vari controller specifici dell'applicazione. La sua finalità principale è fornire strumenti comuni per il recupero dei parametri, la gestione dell'utente autenticato e il coordinamento delle risposte API, in modo che i controller derivati possano concentrarsi sulla logica di business specifica.

Ogni classe controller che estende l'interfaccia controller.py è delegata ai principali e alle funzionalità che sono:

1. Gestione dei parametri della richiesta

- Fornisce utility (es. `self._controller["utils"]`) per estrarre e validare i valori dai parametri della richiesta (`args`) e dagli header (`header`);
- Standardizza la validazione dei parametri e riduce la duplicazione di codice nei controller derivati.

2. Gestione dell'utente autenticato

- Memorizza le informazioni dell'utente loggato in `self._controller["userInfo"]`;
- Permette ai controller derivati di accedere facilmente all'ID utente o ad altri dati rilevanti per autorizzazioni e personalizzazioni.

3. Coordinamento delle risposte API

- Fornisce un formato coerente per le risposte (`response`) contenente almeno lo stato (OK/KO) e il risultato o messaggio associato;
- Gestisce i messaggi standard dell'applicazione tramite `self._controller["msg"]`, centralizzando la gestione delle stringhe di risposta.

4. Supporto alla logica dei controller derivati

- La classe base non implementa logiche specifiche delle entità, ma mette a disposizione gli strumenti per creare metodi CRUD, di condivisione, aggiornamento o rimozione;
- Garantisce che tutte le operazioni sensibili possano fare controllo dei privilegi (es. proprietario della risorsa) in modo uniforme.

5. Strutturazione modulare

- Facilita l'integrazione con i DAO per l'accesso ai dati;
- Permette ai controller derivati di concentrarsi sulla logica applicativa, delegando a Controller le operazioni comuni come parsing dei parametri, validazioni e gestione dell'utente.

Mansioni dei controller

Di seguito vengono riportati i principali metodi di ogni controller

DashboardController.py

- insertNota
- sharedNota
- getNotesSharedByIdUser
- updateNote
- removeNote

UserController.py

- login
- logout
- getAllUsers
- addUser

TokenController.py

- checkTokenAlive

DAO

I DAO (Data Access Object), sono classi dedicate all'interazione con il database. Estendono un'interfaccia base definita in **DaoBase.py**, che riceve in ingresso i dettagli di connessione a uno specifico DBMS. La classe istanzia il **ConnectionManager.py**, il quale utilizza i dati di connessione forniti per creare e restituire l'oggetto di connessione al database.

Ogni classe DAO memorizza le query SQL corrispondenti alle operazioni sul database e fornisce un unico metodo:

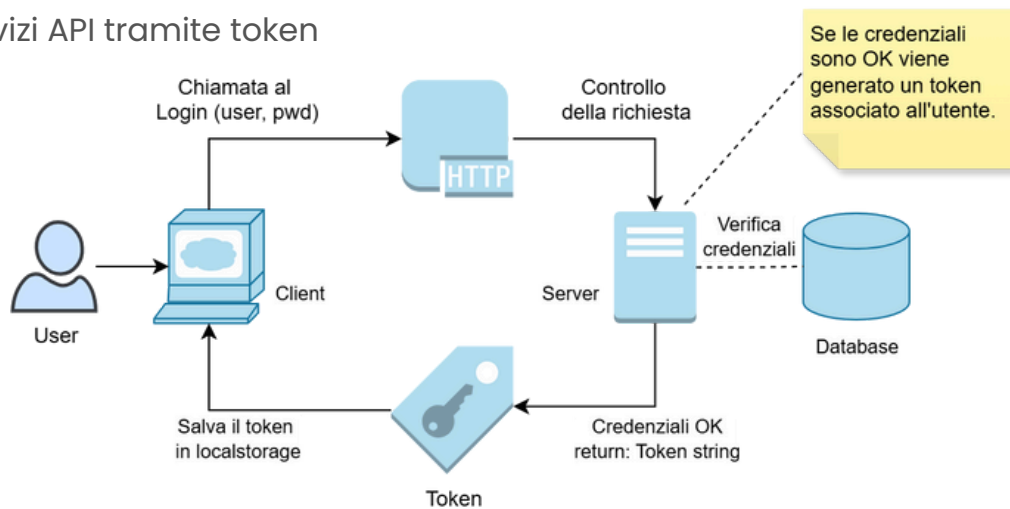
execute(azione, parametri=())

- azione: una stringa che identifica il tipo di operazione CRUD da eseguire su una specifica tabella del database;
- parametri: una lista facoltativa di valori da passare alla query.

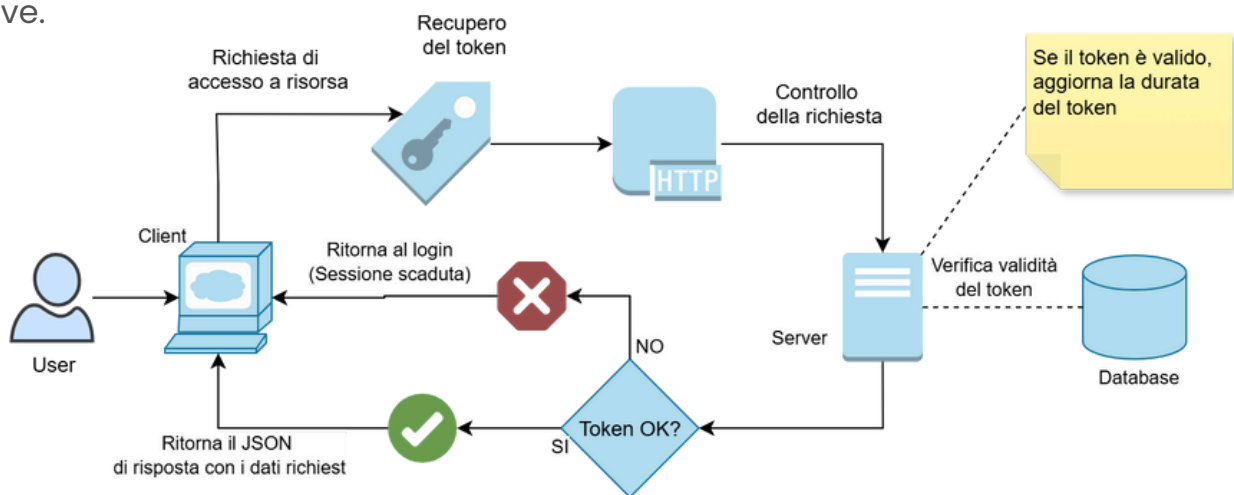
Il metodo execute verifica innanzitutto che l'azione richiesta esista e sia valida. Se la verifica ha esito positivo, esegue la query sul database utilizzando i parametri forniti in modo sicuro, prevenendo vulnerabilità di SQL injection.

Gestione delle sessione - Token

Di seguito viene fornita una breve descrizione di come vengono gestite le sessioni ovvero gli accessi ai servizi API tramite token



Quando un utente effettua il login utilizzando le credenziali (username e password), il backend verifica che l'utente sia registrato. Se la verifica ha esito positivo, viene generato un token, che viene salvato nel database associato all'ID dell'utente e al timestamp di generazione del token. Il token viene poi restituito al client nella risposta di successo, e quest'ultimo lo conserverà nella local session del browser per autenticare le chiamate API successive.



Per le successive richieste HTTP alle API, il frontend recupera il token dalla local session o lo invia nell'header della richiesta. Il backend esegue quindi i seguenti controlli:

1. Verifica che il token esista nel database;
2. Controlla che la differenza tra il timestamp corrente e quello di creazione del token sia inferiore a un valore definito nella configurazione.

Se entrambe le condizioni sono soddisfatte, l'utente può accedere al servizio. Contestualmente, il timestamp del token viene aggiornato con quello della nuova richiesta, estendendo così la validità della sessione.

Configurazione e avvio

Backend

Per avviare il backend bisogna prima configurare alcuni file.

- *connectionsDB.properties*: questo file conserva tutti i dati necessari per accedere al DB ed è così formato:

[notesharingDBMS]**host=127.0.0.1****database=notesharing****port=3306****username=prodUser****password=prodUserNoteSharing2025!****DBMS=mysql**

- *init.properties*: questo file di properties configura l'ambiente API da qui si settano la porta sulla quale verrà startata l'API o in quale configurazione andare a recuperare i dati di accesso al DB si default

[initConfiguration]**version=1.0.0****appName=NoteSharing 1.0****host=0.0.0.0****port=5050****sslContext=false****debuging=true****methodEnabled=GET,POST,PUT,PATCH,DELETE****liveTimeSessionToken=60****resourcingServices=/configFiles/resourcingServices.xml****logFile=/log/logErr.log****baseUrl=http://localhost:4200/****menuCode=DTD****dbDefault=notesharingDBMS**

- *initConfig.ini*: questo file conserva i path relativi a tutti i file di configurazione precedentemente descritti. NOTA: questo file non viene impiegato per questo progetto

Per avviare il back backend bisogna prima assicurarsi di aver installato tramite pip alcune dipendenze:

- flask==3.0.2
- flask-cors==4.0.0
- openai==1.12.0
- pymongo==4.6.1 #non utilizzata per questo progetto
- pymysql==1.1.0
- pyodbc==5.1.0 #non utilizzata per questo progetto
- requests==2.31.0
- mltoDict==0.13.0

Dopo aver verificato che tutto sia stato installato e configurato, per avviare l'API basta spostarsi nella folder dove è presente il file `__init__.py` e avviarlo tramite comando classico: **`python __init__.py`** partirà la seguente schermata di log:



```
LUTETIUM
```

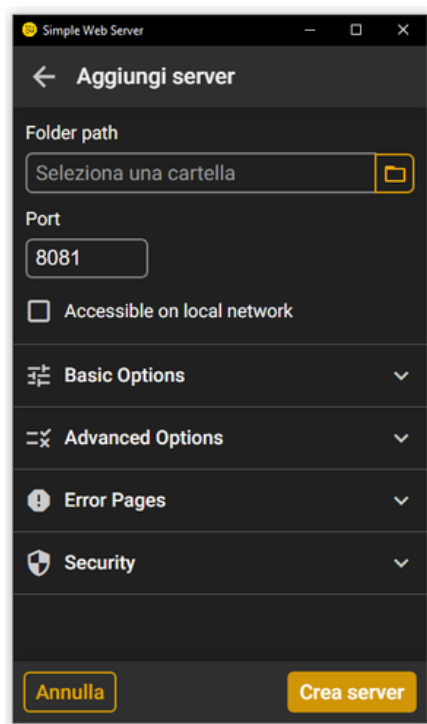
Name	Versions	Date
Lutezio - NoteSharing 1.0	1.0.0	10-09-2025

```
-> Start with  
-> host:0.0.0.0  
-> port:5050  
-> ssl_context:False  
-> debug:True
```

NOTA: l'architettura backend è stata battezzata col nome LUTETIUM (lutezio)

Frontend

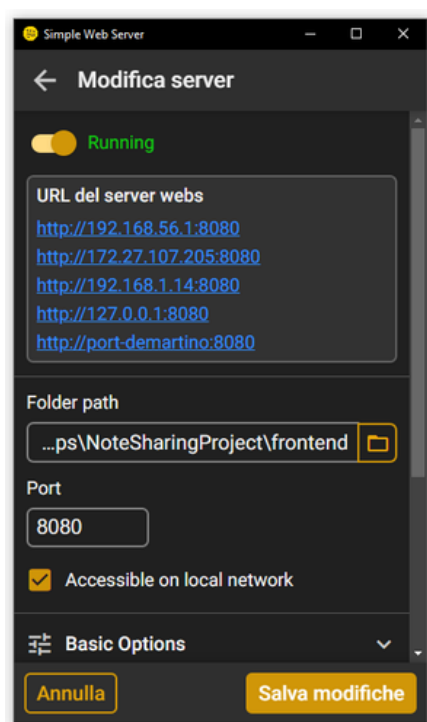
Il front end è un applicazione molto semplice composta da un unico file html: ***index.html***



Per questo progetto è stato utilizzato un server simulator dal quale è possibile simulare un server. Il tool utilizzato si chiama ***Simple Web Server***

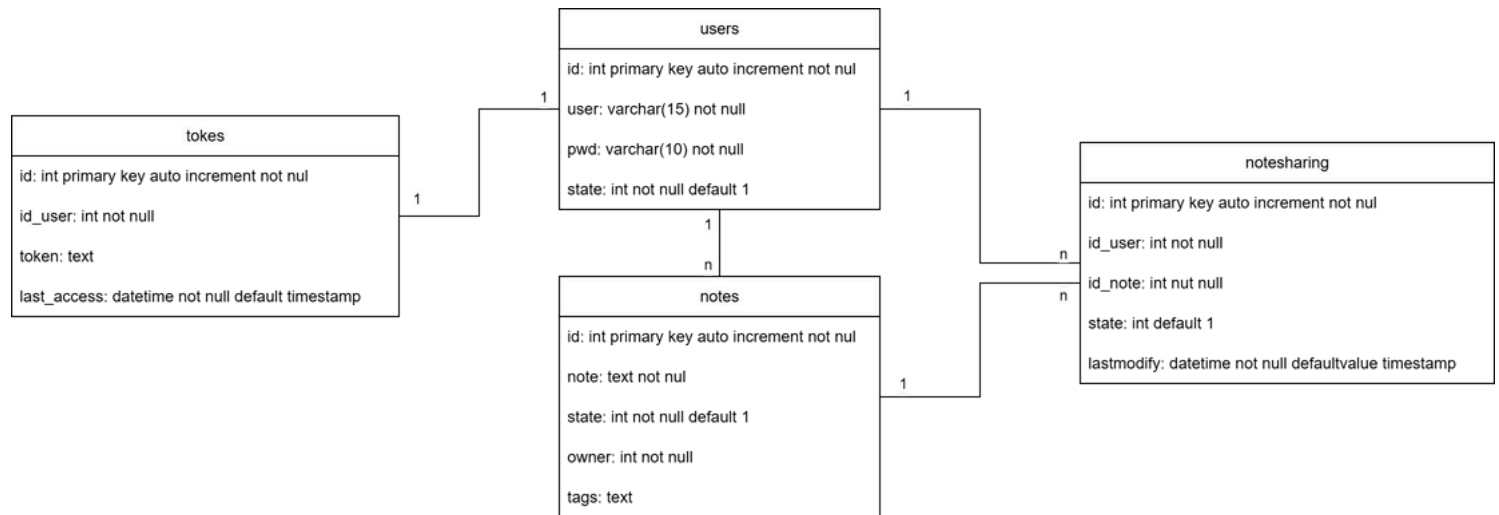
Basta selezionare la folder in cui risiede il codice del front end in particolare la index.html e creare un nuovo server. Il tool avvierà un server virtuale sulla porta 8080 .

NOTA : per risolvere eventuali problemi di CORS origin il backend e il front end avviato tramite Simple Web Server devono avere lo stesso indirizzo IP in questo caso è `http://192.168.1.14` che corrisponde all'IP della macchina locale.



Schema database

Di seguito viene riportato lo schema del db. Per questo progetto è stato utilizzato MySQL, nell'immagine sottostante viene riportato lo schema in class diagram di UML



Sulla repository Git sono stato caricati anche i file necessari per ricreare lo schema in locale