MANUALE SVILUPPATORE - NOTA BENE

Questo manuale vuole essere una guida per guidare nell'avvio del progetto **NotaBene**

Stack utilizzato

Per lo sviluppo del progetto è stato scelto uno stack moderno e bilanciato, che combina stabilità lato backend e flessibilità lato frontend. L'idea è di mantenere il setup semplice ma realistico, in modo da poter riprodurre scenari tipici di un'applicazione enterprise (con API, database e interfaccia web reattiva) senza però complicare troppo l'ambiente di sviluppo.

- **Spring Boot 3 (Java 21)**
- **PostgreSQL 16 (Docker)**
- **React 18**
- **TypeScript (local)**
- **REST API**

Prerequisiti

Per lo sviluppo del progetto è stato scelto un approccio ibrido: backend e database PostgreSQL sono contenuti in docker, mentre il frontend viene eseguito direttamente in console con npm. Per avviare NotaBene, è dunque necessario avere installato sulla propria macchina almeno le seguenti tecnologie:

- **Docker Desktop** (con Docker Compose) ed essersi autenticati all'interno di questo
- **Node.js 16+** e **npm** – per il frontend
- **Git** – per clonare il repository

Nel caso in cui si debbano ancora installare:

- **Git download**: visitare questa pagina e seguire le istruzioni
- **Node download**: visitare questa pagina e seguire le istruzioni
- **Docker Desktop download**: visitare questa pagina e completare l'installazione, aprire docker desktop e registrarsi (operazione molto rapida, si può accedere direttamente anche con il proprio account github o google)

Ottenere il codice

Per ottenere il codice bisogna clonare la repository da Github manualmente scaricandola dal sito oppure tramite i comandi qui sotto eseguirli nel terminale all'interno della cartella dove si vuole salvare NotaBene.

---
git clone <repository-url>
cd NotaBene---SWENG-project-24-25

Configurazioni locali

I file '**.env**' sono presenti sia nella cartella backend che frontend. Sono neccessari per un corretto funzionamento del software. Di solito, questi vengono inclusi nel file '**.gitignore'** in quanto contengono informazioni sensibili. Tuttavia, è stato deciso di includerli nel versionamento del software poichè contengono soltanto dati fittizi utili solo ad avviare l'applicazione. In questo modo, l'avvio è anche più rapido dopo la clonazione della repository in quanto non vi sono files da creare manualmente.

```
# === DATABASE CONFIGURATION ===
POSTGRES_DB=notabenedb
POSTGRES_USER=user
POSTGRES_PASSWORD=userpassword

# === PORTS ===
DB_PORT=5432
APP_PORT=8080

# === SPRING PROFILES ===
SPRING_PROFILES_ACTIVE=docker

# === DATABASE CONNECTION (per connessione diretta dall'host) ===
DB_HOST=localhost
```

Avvio dell'applicazione

Di seguito si illustrano le istruzioni per avviare correttamente tutti i componenti del software.

Avviare il backend

Dopo aver soddisfatto i prerequisiti, spostarsi nella cartella backend ed usare docker compose per buildare ed avviare il container.

```
cd backend
docker compose up --build
```

Attendi nei log la riga di avvio dell'app "Started NotaBeneApplication" che conferma il completo avvio del backend. Eventualmente, controllare anche dall'interfaccia per il desktop se è stato avviato correttamente.

Avviare il frontend

In un altro terminale, spostarsi nella cartella frontend ed installare le dipendenze. Dopodichè, avviare l'applicazione; è sufficiente eseguire i comandi illustrati di seguito.

```
cd frontend
npm install
npm start
```

Accesso rapido

Per l'accesso rapido all'applicazione utilizzare questi link, se la porta del backend è stata modificata nel file .env modificare il link con la porta giusta.

- Frontend: http://localhost:3000
- Backend API: http://localhost:8080/api
- Health check: http://localhost:8080/actuator/health

Chiusura Applicazione

Per chiudere l'applicazione bisogna:

- nel terminale dove si è avviato il frontend fare Ctrl + C
- nel terminale dove si è avviato il backend fare Ctrl + C e poi eseguire il comando `docker compose down`

Altri comandi utili per Docker

```
# Start the backend and database
cd backend
docker compose up --build

# Start in detached mode (background)
docker compose up -d --build

# Stop the backend
docker compose down

# View logs
docker compose logs              # All services
docker compose logs postgres     # Database logs
docker compose logs app          # Backend logs

# Restart the backend
docker compose restart app

# Rebuild without cache
docker compose build --no-cache

# Remove all containers and volumes (⚠️ This will delete all data)
docker compose down -v
```

Struttura del progetto

```
NotaBene---SWENG-project-24-25/
├── README.md
├── start-app.bat                    # Windows script to start backend
├── backend/
│   ├── docker-compose.yml           # Docker configuration (backend only)
│   ├── Dockerfile                   # Docker image for Spring Boot
│   ├── .env                         # ⚠️ CREATE LOCALLY - Environment variables
│   ├── pom.xml                      # Maven configuration
│   ├── init.sql/                    # Database initialization scripts
│   └── src/
│       ├── main/
│       │   ├── java/com/example/myspringapp/
│       │   └── resources/
│       │       ├── application.properties          # Main configuration
│       │       ├── application-docker.properties   # Docker config
│       │       └── application-prod.properties     # Production config
│       └── test/
└── frontend/
    ├── package.json                 # npm configuration
    ├── .env                         # ⚠️ Frontend environment (already created)
    ├── public/
    └── src/
        ├── components/NotesList.tsx
        ├── services/api.ts
        ├── App.tsx
        └── App.css
```

Sicurezza e autenticazione

Il progetto utilizza **Spring Security** per proteggere tutte le API REST, garantendo che solo utenti autenticati possano accedere alle risorse sensibili.

- **Autenticazione tramite token**: ogni richiesta all'API deve includere un token (header **X-Auth-Token**). I token vengono gestiti da **TokenStore**, che mantiene una mappa in memoria tra token e username. Questo permette di validare velocemente le richieste e di associare ciascuna al relativo utente.

- **Filtro di sicurezza personalizzato**: **TokenAuthenticationFilter** intercetta tutte le richieste e verifica il token. Le rotte pubbliche (login, registrazione, monitoraggio **/actuator/health**) sono escluse dal controllo. In caso di token valido, l'utente viene autenticato nel contesto di sicurezza Spring, altrimenti la richiesta viene bloccata con codice 401 Unauthorized.

- **Configurazione centralizzata**: **SecurityConfig** definisce le regole principali, tra cui:

    - disabilitazione del CSRF per le API REST;

    - gestione delle richieste CORS, consentendo origini e metodi specifici (utile per l'integrazione con il frontend React);

    - esposizione dei token tramite header, in modo che il frontend possa leggerli;

    - registrazione del filtro **TokenAuthenticationFilter** nel chain di Spring Security.

- **Password sicure**: le password vengono codificate con **BCrypt**, un algoritmo di hashing robusto, tramite il bean **PasswordEncoder**.

Modulo Controller

Il modulo **controller** espone le API REST dell'applicazione tramite classi annotate con *@RestController*. I controller orchestrano i servizi applicativi, validano gli input e convertono gli esiti in risposte HTTP coerenti, mantenendo la logica di business nei servizi.Ogni controller è mappato sotto un prefisso stabile (*/api/notes*, */api/folders*, */api/tags*, */api/auth*, */api/health*). Le azioni seguono la semantica HTTP: **GET** per letture, **POST** per creazioni, **PUT** per aggiornamenti, **DELETE** per rimozioni. Le risposte sono restituite come *ResponseEntity* con codici di stato consistenti: **201 Created** per creazioni, **204 No Content** per eliminazioni, **400/403/404** per errori previsti. Dove opportuno è supportata la paginazione. Sono disponibili sia una ricerca testuale semplice, sia una ricerca avanzata con filtri per testo, tag, autore e intervalli temporali. La ricerca avanzata accetta sia query string sia payload JSON per scenari più complessi. Le note possono essere condivise gestendo lettori e scrittori tramite endpoint dedicati. I controller applicano i controlli di autorizzazione necessari prima di esporre contenuti o modifiche. Il versionamento delle note consente di visualizzare la cronologia, ispezionare singole versioni, confrontarle e ripristinare lo stato precedente. Le cartelle permettono di organizzare note e di effettuare ricerche contestuali; i tag aiutano a classificare e filtrare in modo trasversale. Un endpoint di *health check* espone lo stato del servizio per l'integrazione con orchestratori e sistemi di monitoraggio. I controller adottano validazione degli input, logging puntuale e gestione degli errori uniforme per rendere le API prevedibili e semplici da integrare.

AuthController **(base path: /api/auth)**
Autenticazione utente: registrazione e login; restituisce un token per le chiamate successive.
• **POST** /register — *register()*
• **POST** /login — *login()*

HealthController **(base path: /api)**
Verifica dello stato dell'applicazione (readiness/liveness).
• **GET** /health — *health()*

NoteController **(base path: /api/notes)**CRUD delle note, ricerca semplice/avanzata, liste rapide e gestione permessi.
• **POST** / — *createNote()*
• **GET** / — *getAllNotes()*
• **GET** /{id} — *getNoteById()*
• **PUT** /{id} — *updateNote()*
• **DELETE** /{id} — *deleteNote()*
• **POST** /{id}/copy — *copyNote()*

• **GET** /search — *searchNotes()*
• **GET** /search/advanced — *searchNotesAdvanced()*
• **GET** /search/advanced-flexible — *searchNotesAdvancedFlexible()*
• **POST** /search/advanced — *searchNotesAdvancedPost()*
• **GET** /created — *getCreatedNotes()*
• **GET** /shared — *getSharedNotes()*
• ... e altre 9 operazioni

NoteVersionController **(base path: /api/notes/{noteId}/versions)**
Storico versioni, dettaglio, confronto e ripristino di una nota.
• **GET** / — *getVersionHistory()*
• **GET** /{versionNumber} — *getVersion()*
• **POST** /{versionNumber}/restore — *compareVersionsEnhanced()*

FolderController **(base path: /api/folders)**
Gestione cartelle, associazione/disassociazione note e ricerche contestuali.
• **GET** / — *listMyFolders()*
• **POST** / — *create()*
• **GET** /{id} — *get()*
• **DELETE** /{id} — *delete()*
• **POST** /{folderId}/notes/{noteId} — *addNote()*
• **DELETE** /{folderId}/notes/{noteId} — *removeNote()*
• **GET** /{folderId}/notes/search — *searchNotesInFolder()*
• **POST** /{folderId}/notes/search — *searchNotesInFolderPost()*
• **GET** /{folderId}/search — *searchInFolder()*
• **POST** /{folderId}/search — *searchInFolderPost()*

TagController **(base path: /api/tags)**
Gestione essenziale dei tag (elenco e creazione).
• **GET** / — *list()*
• **POST** / — *create()*

Modulo DTO

La cartella **dto** definisce il contratto stabile tra backend Spring e frontend React: oggetti semplici e serializzabili (JSON) usati solo ai bordi dell'applicazione (ingresso/uscita dei controller). I DTO proteggono le entità JPA, applicano validazione precoce sugli input e forniscono proiezioni mirate per la UI, evitando leakage di dettagli interni (hash password, lazy fields, ecc.). In pratica, i controller ricevono Request DTO, demandano ai servizi la logica e restituiscono Response DTO già nel formato atteso dal client.

A livello di **comportamento**, il modulo DTO applica tre principi:

1. **Convalida precoce** (Jakarta Validation) → input non valido ⇒ 400 consistente e messaggi chiari;
2. **Disaccoppiamento** dal modello persistente → niente lazy loading né cicli di serializzazione;

3. **Compatibilità evolutiva** → uso di alias Jackson e DTO specifici per casi d'uso (permessi, versioning, diff) per permettere al frontend di evolvere senza toccare le entità.

In pratica, i controller ricevono **Request DTO**, demandano ai servizi la logica e ritornano **Response DTO** già nel formato atteso dalla UI. Questo strato protegge il dominio, evita leakage (es. hash password), uniforma i messaggi di errore e fornisce **payload mirati** (permessi, storico versioni, esiti di confronto) che il frontend può consumare direttamente, mantenendo l'API pulita e stabile.

Modulo Entity

La cartella **entity** contiene il modello di dominio dell'applicazione: le classi annotate JPA che mappano gli oggetti di business alle tabelle PostgreSQL. Qui vivono gli oggetti "veri" che il resto del sistema manipola (es. utenti, note, versioni delle note) ed è da queste entità che i repository Spring Data JPA leggono e scrivono sul database. In generale, ogni classe definisce: una chiave primaria (tipicamente Long), eventuali vincoli di validazione (Bean Validation), relazioni tra oggetti (es. @ManyToOne tra Note e il suo proprietario) e campi temporali (creazione/aggiornamento) utili per l'audit applicativo.

Per la collaborazione sulle note, il dominio modella sia la proprietà sia la condivisione: la Note ha un proprietario/creatore e mantiene due liste di autorizzazioni logiche (lettori e scrittori). Queste raccolgono gli ID utenteabilitati alla lettura/scrittura; vengono persistite come liste scalari (via @ElementCollection o tipo JSON a seconda della configurazione) e sono usate dal servizio applicativo per applicare le regole di accesso. Un flusso tipico: quando un utente "lascia" una nota condivisa, il suo ID viene rimosso da entrambe le liste (readers/writers), riflettendo lo stato corrente dei permessi direttamente a livello di entità.

La versionatura dei contenuti è modellata con un'entità dedicata (p.es. NoteVersion): ogni modifica rilevante produce un record immutabile con riferimento alla nota, numero di versione, snapshot del contenuto e timestamp (ed eventualmente l'autore della modifica). Questo consente di costruire rapidamente lo storico e di esporre DTO leggeri verso il frontend senza esporre direttamente le entità.

Dal punto di vista architetturale, le entità sono puramente dati: non contengono logica di sicurezza (che è nel filtro/token) né regole di orchestrazione HTTP. La sicurezza applicativa si appoggia ai dati presenti nelle entità (owner, liste di accesso, ecc.) ma le decisioni sono prese nei servizi/controller. Per evitare cicli di serializzazione e leakage di dettagli interni, l'esposizione al client avviene tramite DTO e mapper dedicati; nelle entità si usano, dove serve, annotazioni Jackson (@JsonIgnore, ecc.) solo per casi puntuali.

Modulo Exception

La cartella **exception** centralizza la gestione degli errori API con @RestControllerAdvice e mappa le eccezioni a status HTTP coerenti restituendo un payload uniforme. Tutte le risposte usano ErrorResponse { message, status, timestamp, errors[] } per dare un formato stabile alla UI (timestamp via LocalDateTime.now(); errors con dettagli opzionali).

- Mappature principali:

- NoteNotFoundException → 404 Not Found (risorsa inesistente);
- UnauthorizedNoteAccessException → 403 Forbidden (nota non accessibile all'utente);
- DataIntegrityViolationException → 409 Conflict (vincoli DB: duplicati/violazioni);
- MethodArgumentNotValidException → 400 Bad Request (input non valido con lista errori);
- Exception generica → 500 Internal Server Error (messaggio sintetico, no stack trace).

Un beneficio che porta questo modulo è quello di separare la logica di errore dai controller, garantisce codici e messaggi consistenti, semplifica i test e rende il frontend indipendente dai dettagli interni (eccezioni, DB, sicurezza).

Modulo Model

La cartella **model** contiene le entità JPA "core" e riusabili dell'applicazione insieme ai value object che supportano chiavi composte. Qui risiedono i componenti di dominio trasversali (non specifici di una singola feature), mappati su PostgreSQL e consumati dai repository e dai servizi; l'esposizione al client avviene sempre tramite DTO. Il package model definisce lo scheletro persistente condiviso (utenti, tag, chiavi composte), mantenendo le entità povere di logica e focalizzate sulla mappatura. Le policy (autorizzazioni, regole di condivisione, validazioni business) restano nei servizi; i controller parlano in DTO per evitare leakage (es. password) e vincoli di JPA verso il frontend.

Modulo Repository

La cartella **repository** espone l'interfaccia di accesso al database tramite Spring Data JPA, incapsulando sia le CRUD standard sia query mirate sulle feature chiave (permessi, ricerca, versioning, cartelle/tag). I repository sono stateless, utilizzati dai service per comporre use case, e restituiscono entità già pronte per la proiezione in DTO.

Per esempio per note e permessi (PostgreSQL arrays) — NoteRepository modella l'accesso permission-based usando query native con ANY() su array readers/writers (es. *findByReadersContaining*, *findByWritersContaining*) e include metodi di controllo puntuale (lettura/scrittura su una nota specifica). Supporta ricerca full-text semplice(titolo/contenuto) e una ricerca avanzata con join su tag, users (autore) e folder_notes, più filtri opzionali su autore, intervalli temporali (created/updated), folderId, con ordinamento per data e varianti paginabili per la lista note.

Come beneficio le regole di sicurezza rimangono nei service/controller, mentre il package repository garantisce query efficienti e coese, riutilizzabili e testabili, con chiara separazione tra persistenza e contratto API(DTO).

Modulo Service

Nel modulo service vive il "cervello" dell'applicazione: qui si decidono i casi d'uso, si applicano le regole di permesso e si orchestrano repository e DTO. I controller restano

sottili, perché tutto ciò che riguarda cosa è consentito, come validare i dati e come comporre la risposta per la UI accade a questo livello, dentro confini transazionali chiari.

Il cuore funzionale è la gestione delle note. Il servizio dedicato crea, aggiorna, elimina e anche duplica le note, occupandosi prima di associare correttamente tag e metadati e poi di salvare. La parte più importante è la sicurezza applicativa: ogni operazione verifica se l'utente è proprietario o comunque abilitato (reader/writer) e, in caso contrario, solleva eccezioni mirate che lo strato *exception* traduce in HTTP coerenti. La condivisione è gestita in modo esplicito: aggiunta/rimozione permessi per ID o username, e la possibilità per un utente di uscire da una nota condivisa rimuovendosi da entrambi gli elenchi. Per la ricerca, lo stesso servizio espone sia una modalità veloce (titolo/contenuto filtrati per permessi), sia una ricerca avanzata che combina query testuale, autore, tag, intervalli temporali e, quando serve, il contesto di una cartella. L'output non sono entità JPA, ma DTO pronti per la UI, completi di flag come isOwner o canEdit.

Ogni modifica significativa a una nota è versionata in automatico. Un servizio dedicato cattura lo stato precedente prima dell'aggiornamento, costruisce uno storico ordinato e permette il ripristino puntuale (titolo, contenuto e perfino i permessi tornano a come erano). Per rendere l'esperienza fluida, lo storico include anche la "versione corrente" come se fosse una versione a sé: così il frontend può confrontare sempre due stati. A supporto, un servizio di diff testuale calcola le differenze a livello di caratteri e le riconsegna come segmenti EQUAL/ADDED/REMOVED, già pronti per essere evidenziati in UI senza ricalcoli lato client.

Le funzionalità di tag e cartelle completano il quadro. I tag vengono creati con gestione dei duplicati e restituiti in liste ordinate, utili per autosuggest e filtri. Le cartelle offrono un CRUD focalizzato sull'utente: si può vedere il dettaglio con le note collegate e gestire l'associazione nota↔cartella con controlli rigorosi (la cartella deve essere tua, la nota deve appartenerti, niente doppioni), così l'esperienza rimane coerente e prevedibile.

Trasversalmente, il modulo adotta alcune regole semplici ma fondamentali: metodi @Transactional (read-only quando possibile) per coerenza e performance, controlli di autorizzazione nel service e non nei controller, eccezioni esplicite per i casi d'errore, log essenziale con i dati utili alla diagnosi, e conversione sistematica a DTO per consegnare al frontend payload stabili, sicuri e adatti al rendering. In breve: i *service* trasformano le intenzioni dell'utente in azioni corrette e verificabili, tenendo insieme sicurezza, dati e UX.

All'interno del modulo service vi è anche il modulo **Support** e il modulo **Memento**

Support

Il modulo **support** raccoglie piccoli componenti iniettabili che fanno da ponte tra contesto di sicurezza e persistenza, così i service restano puliti e focalizzati sui casi d'uso

In sintesi, il package support incapsula la lettura del principal e i controlli di ownership in componenti riutilizzabili: meno codice ripetuto nei service, errori 401 coerenti quando manca l'identità, e verifiche di possesso efficienti lato DB quando servono decisioni di sicurezza.

Memento

Il modulo memento implementa il pattern GoF *Memento* per catturare e ripristinare lo stato completo di una nota senza esporre dettagli interni ai servizi. L'idea è semplice: prima di ogni modifica significativa, creiamo un istantanea immutabile della nota; se qualcosa va storto o l'utente vuole tornare indietro, ripartiamo da quella istantanea invece di ricostruire manualmente i campi.

Frontend

Il frontend è una SPA React 18 + TypeScript organizzata per feature component con servizi HTTP centralizzati. La regola è semplice: i componenti presentano la UI e orchestrano piccoli stati locali; tutte le chiamate passano da services/api, che espone tipizzazioni (Note, TagDTO, ecc.) e funzioni per note, cartelle, versioni e autenticazione. Lo stile è gestito con CSS per-componente (file .css affiancati), senza dipendenze da framework UI pesanti.

La home mostra un benvenuto contestuale (login/non login) e, se autenticato, la preview dell'ultima nota, ricavata ordinando per createdAt, i pulsanti rimandano alla vista note. Le azioni (Accedi/Registrati/Logout) sono demandate a callback passate dal contenitore principale, coerenti con un routing leggero a stato locale.

Autenticazione. Login e registrazione sono form controllati con validazione minima lato client e gestione degli errori HTTP (401/403/409) per messaggi chiari. Dopo login/registrazione, il token viene salvato via authApi, mentre l'UI memorizza lo username in localStorage per un uso rapido nell'interfaccia.

La vista principale della pagina delle note elenca le note di proprietà e quelle condivise, separandole visivamente e abilitando azioni in base ai flag (isOwner, canEdit, canDelete, canShare) ricevuti dal backend. Da qui si può modificare, copiare, eliminare (con conferma modale), uscire da una nota condivisa e gestire i permessi tramite una modale dedicata. La ricerca è doppia: semplice (query pieno testo) e avanzata (filtri combinati), entrambe possono essere circoscritte alla cartella selezionata. Le conferme/avvisi usano un hook modale condiviso, mentre le chiamate passano da notesApi/foldersApi.

Creazione e modifica note. I form "Crea" e "Modifica" applicano le stesse regole di lunghezza (titolo ≤ 255, contenuto ≤ 280), mostrano i char counter e integrano la selezione dei tag. I tag selezionati sono inviati come tagIds al backend; la UI offre anche la creazione rapida di un nuovo tag tramite modale, aggiornando lo stato locale senza duplicati. Gli errori provenienti dall'API sono normalizzati in messaggi leggibili (anche quando l'API restituisce array di errori).

Gestione tag (modale). La creazione di un tag avviene in portal con overlay cliccabile per chiusura, validazione immediata e normalizzazione robusta degli errori (stringa, oggetto {message}, array errors, ecc.), garantendo un feedback coerente indipendentemente dalla forma della risposta server.

Cartelle (sidebar). La sidebar carica l'elenco cartelle dell'utente, consente selezione (filtra la lista note), creazione inline e cancellazione con conferma; cura dettagli di accessibilità (tasti Invio/Spazio, aria-current) e segnala gli stati di caricamento/errore. L'integrazione con la lista note permette azioni "aggiungi/rimuovi dalla cartella" direttamente dalle card quando una cartella è attiva.

La modale “Versioni” carica lo storico della nota, permette di visualizzare i dettagli di una versione, switchare la versione attiva (senza creare nuove versioni) e lanciare un confronto avanzato tra due versioni, che apre un viewer dedicato con evidenziazione dei segmenti aggiunti/rimossi/uguali. La versione attiva è evidenziata e lo switch richiama il servizio di ripristino; la chiusura modale e il refresh della lista sono gestiti via callback.

Pattern e principi adottati

Information Expert (GRASP)

Nel progetto le responsabilità sono assegnate a chi possiede già tutte le informazioni utili: NoteService governa permessi e ricerca perché conosce utente corrente, nota e query dei repository, restituendo direttamente NoteResponse con flag come isOwner e canEdit; NoteVersioningService custodisce la storia delle versioni e le regole di ripristino; TextDiffService incapsula l’algoritmo di diff; CurrentUserResolver è l’esperto dell’identità. Questo evita passaggi inutili, riduce duplicazioni e rende i flussi più lineari.

Creator (GRASP)

La creazione degli oggetti avviene dove ha più senso: i service istanziano le entità/DTO che poi usano o memorizzano (NoteService crea Note, NoteVersioningService crea NoteVersion, TagService crea Tag), mentre i DTO espongono factory statiche come NoteResponse.fromEntity(...). Così l’inizializzazione resta coerente e completa nel punto che possiede i dati necessari, diminuendo gli errori di costruzione dispersi.

Controller (GRASP)

I controller REST ricevono le “system operation” (HTTP), validano i DTO e delegano la logica ai service, mantenendo sottile lo strato d’ingresso. In parallelo, nel frontend, il modulo services/api funge da intermediario tipizzato tra componenti e backend: l’interfaccia utente rimane pulita, la logica applicativa è concentrata e i test diventano più semplici.

Low Coupling (GRASP)

Il disaccoppiamento è ottenuto con interfacce e iniezione delle dipendenze: CurrentUserResolver e NoteOwnershipChecker mascherano dettagli di sicurezza/DB, i repository Spring Data incapsulano la persistenza e i DTO schermano il dominio dall’API. Il risultato è un impatto minimo ai cambi, mock facili nei test e una struttura che regge bene l’evoluzione.

High Cohesion (GRASP)

Ogni classe fa “una cosa sola, ma bene”: NoteService si occupa di note/permessi/ricerca, NoteVersioningService solo di versioni, TagService di tag, FolderService di cartelle; il modulo exception standardizza gli errori, il modulo memento gestisce snapshot e ripristino. La coesione alta rende il codice leggibile, prevedibile e manutenibile.

Pure Fabrication (GRASP)

Quando una responsabilità non appartiene in modo naturale al dominio, viene estratta in componenti tecnici riusabili: TextDiffService per il diff, CurrentUserResolver e NoteOwnershipChecker per identità/ownership, GlobalExceptionHandler e ApiExceptionHandler per il contratto d'errore. Queste "fabbricazioni pure" migliorano coesione e riducono accoppiamento senza appesantire le entità.

Indirection (GRASP)

Per disaccoppiare chi invoca da chi esegue, il progetto inserisce mediatori: i resolver/checker JDBC stanno tra service e DB quando serve solo un boolean; services/api nel frontend sta tra UI e HTTP; la security filter chain media tra HTTP e controller. Questo strato intermedio stabilizza le dipendenze e offre punti di estensione chiari.

Polymorphism (GRASP)

I service dipendono da astrazioni, non da implementazioni concrete: *Resolver e *Checker possono essere sostituiti (JDBC, mock di test, implementazioni future) senza toccare i client; anche i repository sono proxati da Spring. Il polimorfismo permette di estendere comportamenti senza riscrivere chi li usa.

Protected Variations (GRASP)

Le parti soggette a variazione sono protette da interfacce e contratti stabili: i DTO isolano il client dallo schema dati, i service nascondono dettagli del token store o della modellazione dei permessi, i componenti di supporto espongono API minimali. Così si rispettano OCP e si riduce il rischio che un cambiamento interno rompa l'esterno.

Strategy (GoF)

CurrentUserResolver/JdbcCurrentUserResolver e NoteOwnershipChecker/JdbcNoteOwnershipChecker rappresentano famiglie di algoritmi intercambiabili dietro un'interfaccia. I service invocano il "cosa" senza conoscere il "come": in produzione usi JDBC, nei test sostituisci con stub. Aggiungere una strategia non richiede modifiche ai consumatori.

Template Method (GoF)

TokenAuthenticationFilter implementa doFilterInternal(...) all'interno dello scheletro offerto da OncePerRequestFilter. Spring definisce il flusso generale del filtro e il progetto fornisce solo i passi variabili (estrazione/validazione token, set dell'autenticazione). Si ottiene riuso dell'algoritmo comune con la massima personalizzazione dove serve.

Memento (GoF)

NoteMemento cattura lo stato di una nota senza violarne l'incapsulamento e NoteVersionManager gestisce la vita dei memento; NoteVersioningService li usa per storico e ripristino. Ogni modifica significativa produce uno snapshot immutabile e il ripristino è atomico e sicuro, con grandi benefici su affidabilità, audit e UX.

Repository / DAO (Pattern architetturale)

I vari *Repository incapsulano persistenza e query (anche native), offrono metodi chiari per ricerche e controlli di permesso e restituiscono entità pronte per i service. Così la logica di business non “vede” SQL o mapping, i test isolano la persistenza e il cambio di strategie di query non impatta i consumer.

Facade (GoF)

Nel frontend, services/api è una facciata unica e tipizzata per tutte le chiamate: login, note, versioni, cartelle, tag. I componenti restano presentazionali e se cambia un header o un endpoint, si interviene in un solo posto. Il beneficio è una UI più semplice e un minor accoppiamento con il backend.

Chain of Responsibility (GoF)

La catena dei filtri di Spring Security, che include TokenAuthenticationFilter, processa la richiesta in step indipendenti: estrazione token, validazione, popolamento del contesto, proseguimento. L’autenticazione diventa un concern trasversale componibile e riordinabile, senza logica di sicurezza nei controller.

Factory Method (idioma)

Le factory statiche nei DTO (fromEntity, of, ecc.) centralizzano la creazione delle proiezioni verso la UI. Il codice chiamante resta pulito, la costruzione è auto-documentata e l’API può evolvere senza riflessi su chi istanzia gli oggetti.

Value Object (DDD/Mapping)

FolderNoteId come @Embeddable rappresenta la chiave composta nota↔cartella con equals/hashCode corretti e semantica esplicita. Il mapping è più sicuro, i set/map funzionano come previsto e si riducono i bug su chiavi composite.

Proxy e Singleton (di container)

Molti bean sono gestiti come singleton e proxati da Spring per abilitare transazioni e AOP senza codice boilerplate. Il lifecycle è uniforme, i cross-cutting concern restano fuori dalla logica applicativa e i service possono concentrarsi sui casi d’uso.

Test

Backend (JUnit + Spring Test)

I test presenti sono in: backend/src/test/java/...

Esecuzione locale:

```
cd backend
mvn test
# oppure, se presente il wrapper
./mvnw test
```

Nel progetto sono presenti sia unit test mirati sia integration test end-to-end. Gli unit test, ad es. NoteControllerTest con @WebMvcTest, isolano il controller: il NoteService è mockato (@MockBean), la security è esclusa, e si verifica il contratto HTTP/JSON e il wiring (creazione nota 201, validazioni 400 con payload d'errore uniforme, 404 su NoteNotFoundException, corretta delega ai metodi del service, paginazione che invoca getAllNotesPaginated(...), esiti degli endpoint di *leave* con 200/400/404 in base alle eccezioni). Gli integration test, ad es. NoteIntegrationTest con @SpringBootTest + @AutoConfigureMockMvc, esercitano lo stack reale (controller, service, repository, security, GlobalExceptionHandler), popolano il DB con UserRepository/NoteRepository, generano token in TokenStore e chiamano gli endpoint con gli header (X-Auth-Token, Authorization) verificando l'intero flusso: CRUD note (201/200/204), ricerca, permessi (flag isOwner/canEdit/canDelete/canShare per owner/reader/writer), gestione permessi (add/remove readers/writers con effetti osservabili), leave che rimuove l'utente da entrambe le liste, e i casi di sicurezza (401 senza token o token invalido, 403 quando un non-owner tenta di gestire i permessi, 404 per note non visibili). Insieme, i due livelli garantiscono feedback rapido e preciso: gli unit test bloccano regressioni del contratto API e della validazione, mentre gli integration test assicurano che tutto il sistema inclusi sicurezza, persistenza e mapping DTO funzioni davvero come previsto.

Note importanti per i test dei Controller

- Gli endpoint (tranne /api/auth/** e /actuator/health) richiedono token.
  Nei test con MockMvc, imposta l'header X-Auth-Token con un token noto allo store oppure configura il contesto di sicurezza per bypassare il filtro.
- Se nei test vedi 403/401, controlla:
  - Presenza header X-Auth-Token coerente con lo store usato nel test.
  - Import/registrazione del filtro TokenAuthenticationFilter quando serve, o un suo mock @MockBean se vuoi isolarne gli effetti.
  - Se usi @WebMvcTest, valuta addFilters = false o config mirata alle classi di sicurezza.

Troubleshooting

Backend non parte (porta occupata)
Modifica porte in backend/.env, es.: APP_PORT=8081, DB_PORT=5433 e riavvia.

Variabili d'ambiente non lette
Controlla che backend/.env esista, senza spazi attorno a = e con encoding UTF-8.

DB non raggiungibile
docker compose ps → verifica container; docker compose logs postgres → controlla credenziali.

Frontend non raggiunge l'API
curl http://localhost:8080/actuator/health → backend UP.
Controlla frontend/.env (REACT_APP_API_URL), CORS e il token.

Chiamate API → 403/401
Aggiungi X-Auth-Token (il frontend lo fa via interceptor se il token è salvato). Nei test MockMvc ricordati di impostare l'header.

Build Docker fallisce
Assicurati che Docker Desktop sia attivo; prova docker compose build --no-cache; verifica spazio disco.

Reset completo backend

```
# Stop and remove everything (⚠️ This deletes all data)
cd backend
docker compose down -v

# Start fresh
docker compose up --build
```