Manuale dello Sviluppatore – NOTA BENE

1. Panoramica del progetto

NOTA BENE è un progetto universitario di Ingegneria del Software sviluppato come applicazione web per creare, organizzare e condividere note testuali(≤ 280 caratteri) con versionamento e permessi di lettura/scrittura. La soluzione si basa su uno stack che comprende Spring Boot per il backend, HTML/CSS/JavaScript per il frontend, PostgreSQL ospitato su Supabase e containerizzazione con Docker/Compose, con il supporto di test automatizzati tramite JUnit. Il repository include una ricca documentazione e una struttura di sorgenti ben organizzata in controller, servizi, modelli, DTO e relative directory di test per controller, servizi, repository e integrazione.

2. Requisiti e installazione

Prerequisiti:
- Java 17+
- Maven 3.6+
- Docker & Docker Compose
- PostgreSQL (Supabase)

Clonazione repository:
git clone https://github.com/franksToscani/nota-bene.git
cd nota-bene

Configurazione database:
host: aws-0-eu-central-2.pooler.supabase.com
port: 5432
database: postgres
user: postgres.ijdurgjffeyrxdwhrqcx
pool_mode: transaction

3. Esecuzione e test

- ➔ Build: *mvn clean package*
- ➔ Avvio con Docker: *docker-compose up --build*
- ➔ Avvio locale: *mvn spring-boot:run*
- ➔ Test: *mvn test*
- ➔ Avvio senza test: *mvn -DskipTests package && mvn spring-boot:run -DskipTests*

4. Architettura e struttura del codice

Struttura per layer:
src/main/java/com/sweng/nota_bene/
├── controller/ # REST controller
├── service/ # logica applicativa
├── repository/ # accesso dati
├── model/ # entità JPA
├── dto/ # Data Transfer Objects
└── config/ # configurazioni (es. sicurezza)

5. Pattern GoF – Definizioni, uso ed esempi

Singleton

Garantisce una sola istanza con accesso globale.

Nel progetto:

- Dove: `SecurityConfig` definisce tre metodi `@Bean` (`securityFilterChain`, `passwordEncoder`, `customLogoutSuccessHandler`) che Spring istanzia una sola volta per tutta l'applicazione
- Come & perché: la configurazione centralizzata richiede risorse condivise (encoder, filtri e handler), garantendo un unico oggetto riutilizzato e semplificando la gestione della sicurezza.

Esempio di codice:

```
@Bean
public PasswordEncoder passwordEncoder() {
 return new BCryptPasswordEncoder();
}
```

Strategy

Famiglia di algoritmi intercambiabili.

Nel progetto:

- Dove: nello stesso `SecurityConfig`, il `LogoutSuccessHandler` è fornito come lambda personalizzabile, e `PasswordEncoder` è scelto come implementazione concreta (`BCryptPasswordEncoder`) di un'interfaccia comune
- Come & perché: le strategie di logout e di hashing possono essere sostituite facilmente con altre implementazioni, disaccoppiando l'algoritmo dal codice che lo utilizza.

Esempio di codice:

```
@Bean
public LogoutSuccessHandler customLogoutSuccessHandler() {
 return (request, response, authentication) -> {
 response.setStatus(200);
 response.setContentType("application/json");
 response.getWriter().write("{\"success\": true, \"message\":
\"Logout effettuato con successo\"}");
 };
}
```

Template Method

Definisce scheletro algoritmo con hook.

Nel progetto:

- Dove: nel modello `Note`, i metodi annotati `@PrePersist` e `@PreUpdate` forniscono hook che JPA chiama prima del salvataggio/aggiornamento
- Come & perché: la logica standard di creazione/aggiornamento è definita nella superclasse JPA, mentre le sottoclassi (qui la nostra entità) personalizzano alcuni passi, come impostare date automatiche.

Esempio di codice:

```
@PrePersist
protected void onCreate() {
 dataCreazione = OffsetDateTime.now(ZoneOffset.UTC);
 dataUltimaModifica = dataCreazione;
}
```

Composite

Composizione di specifiche.

Nel progetto:

- Dove: `NoteSpecification` crea piccole specifiche (`withOwnerOrShared`, `withSearchTerm`, etc.) che possono essere composte e concatenate; `NoteService` le unisce con `.and()` per formare query complesse
- Come & perché: ogni specifica rappresenta un criterio di filtraggio; combinandole si costruisce un albero di oggetti che il repository interpreta come un'unica query, facilitando l'estensione e la riusabilità dei filtri.

Esempio di codice:

```
var spec =
NoteSpecification.withOwnerOrShared(proprietarioEmail,
idsCondivisi)
 .and(NoteSpecification.withSearchTerm(searchTerm))
 .and(NoteSpecification.withTag(tag));
```

Factory Method

Metodo dedicato per la creazione di oggetti.

Nel progetto:

- Dove: `TagResponse.from` fornisce un metodo statico che crea l'oggetto e viene usato da `TagService.getAllTags` per trasformare ogni entità `Tag` in DTO
- Come & perché: centralizza la logica di creazione delle risposte sui tag e permette di modificare in futuro il processo d'istanza senza impattare il codice client.

Esempio di codice:

```
public record TagResponse(String nome) {
 public static TagResponse from(String nome) {
 return new TagResponse(nome);
 }
}
```

Adapter

Traduce interfaccia in un'altra attesa.

Nel progetto:

- Dove: i metodi privati `mapToNoteResponse` e `mapToNoteListResponse` di `NoteService` convertono un'entità `Note` in differenti DTO per l'API
- Come & perché: adattano il modello dati interno a un formato più semplice e stabile per i consumer, isolando controller e client dai dettagli dell'entità persistente.

Esempio di codice:

```
private NoteResponse mapToNoteResponse(Note note) {
 return new NoteResponse(
 note.getId(), note.getTitolo(), note.getContenuto(),
 note.getProprietario(), note.getDataCreazione(),
 note.getDataUltimaModifica(), note.getIdCartella(),
 note.getTag(),
condivisioneService.getCondivisioniByNota(note.getId())
 );
}
```

6. Autenticazione e gestione sessioni

- Registrazione e login: inviano JSON a `/api/auth/register` e `/api/auth/login`, con autenticazione automatica dopo la registrazione
- Form grafici: login con nickname e password, registrazione con email, nickname e password
- Verifica sessione e logout: `GET /api/auth/check` restituisce i dati dell'utente; `POST /api/auth/logout` invalida la sessione
- Sicurezza: Spring Security disabilita CSRF, permette l'accesso anonimo solo a pagine statiche e auth API, forza una sola sessione attiva e gestisce il logout con cancellazione del cookie
- Header dinamico: mostra iniziali e nickname dell'utente e offre il pulsante "Esci"

7. Funzionalità principali

1) **Home page**
    a) Ricerca e filtri: barra di ricerca, filtri per data di creazione e modifica, contatore risultati e pulsanti "Filtra/Azzera"
    b) Azioni sulle note: ogni nota ha menu con *Modifica*, *Duplica* e *Elimina*; clic sul corpo espande/chiude il testo
    c) Vista per cartelle: pulsante che alterna lista e raggruppamento per cartelle, con caricamento dinamico di note e cartelle dell'utente

2) **Visualizzazione nota**
    a) Cliccando su una nota si apre un modal che mostra titolo e contenuto completo, con possibilità di chiusura cliccando fuori o su “×”

3) **Creazione e modifica di una nota**
    a) Campi obbligatori e limiti: titolo ≤255 caratteri e contenuto ≤280 caratteri
    b) Form grafico: titolo, tag, contenuto con contatore, sezione condivisioni, selezione/creazione cartella, pulsanti Annulla e Salva
    c) Validazioni in tempo reale e salvataggio: il frontend aggiorna il contatore e invia i dati via `POST /api/note` (creazione) o `PUT /api/note/{id}` (modifica)

4) **Tag**
    a) Elenco tag recuperato da `GET /api/tag`; l’app li ordina alfabeticamente e li crea se non esistono già

5) **Condivisione e permessi**
    a) Nel form è possibile aggiungere email con permesso di *lettura* o *scrittura*
    b) Il backend impedisce l’auto-condivisione e verifica che gli utenti esistano prima di salvare i permessi
    c) Ogni condivisione include tipo (`lettura` o `scrittura`) validato lato API

6) **Gestione cartelle**
    a) Recupero cartelle dell’utente e creazione di nuove tramite modal: `GET/POST /api/cartelle`

7) **Ricerca avanzata**
    a) L’interfaccia utente della pagina principale mette a disposizione due insiemi di campi data, uno per l’intervallo di **creazione** e uno per quello di **modifica**, accompagnati dai pulsanti *Filtra* e *Azzera* per applicare o resettare i filtri. Quando l’utente preme *Filtra*, la funzione `applyFilters` legge i valori inseriti, li normalizza nel formato `YYYY-MM-DDT00:00:00Z` e li invia come querystring alla richiesta `GET /api/note/search`. Il controller, a sua volta, espone l’endpoint `/api/note/search`, accetta i parametri opzionali (`createdFrom`, `createdTo`, `modifiedFrom`, `modifiedTo`) in formato ISO 8601 e li inoltra al servizio, convertendo le stringhe vuote in `null`. Nel backend, il `NoteService` costruisce una **Specification JPA** che, oltre ai filtri su proprietario, testo e tag, applica condizioni sulle date di creazione e modifica utilizzando i metodi `withCreationDateBetween` e `withLastModifiedDateBetween`

definiti in `NoteSpecification`. L'entità `Note` gestisce i timestamp tramite gli attributi `dataCreazione` e `dataUltimaModifica`: il primo viene impostato automaticamente alla creazione, mentre il secondo si aggiorna a ogni modifica grazie ai metodi annotati con `@PrePersist` e `@PreUpdate`.

8) **Operazioni sulla nota**
    a) Duplicazione: endpoint `/api/note/{id}/copy` e voce di menu *Duplica* creano una copia nella propria raccolta
    b) Eliminazione: solo il proprietario può cancellare la nota

8. Modello dati

Nota: id (UUID), titolo (≤255), contenuto (≤280), proprietario (email), date di creazione/modifica, cartella e tag opzionali

9. Sicurezza e gestione errori

Accesso autenticato alle API
Password hashing BCrypt
Gestione errori centralizzata con GlobalExceptionHandler

10. Tipi di test

- Unit test: JUnit 5 + Mockito
- Web layer: @WebMvcTest + MockMvc
- Integration test: @SpringBootTest + Testcontainers

11. Contributi

- Fork del repository e creazione di un branch dedicato.
- Implementazione delle modifiche seguendo la struttura esistente.
- Esecuzione dei test (mvn test) e build (mvn clean package).
- Apertura di una pull request descrivendo le modifiche.