



UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Scriptoria

Autori:

Cristiano Acciai & Niccolò Bartoli

Corso:

Ingegneria del software

N° Matricola:

7076811 & 7076612

Docente:

Enrico Vicario

Indice

1	Introduzione	4
1.1	Descrizione del progetto	4
1.2	Struttura del progetto e ambiente di sviluppo	4
2	Progettazione del software	5
2.1	Attori e casi d'uso	5
2.1.1	Utente predefinito	7
2.1.2	Autore	7
2.1.3	Moderatore	10
2.1.4	Admin	11
2.2	Mockups	12
2.3	Class diagram	16
2.3.1	Design pattern: Singleton	18
2.3.2	Design pattern: Mapper	19
2.3.3	Design pattern: Builder	19
2.3.4	Design pattern: DAO	21
2.4	Diagramma ER e Modello Relazionale	21
3	Implementazione delle classi	23
3.1	Business Logic	23
3.1.1	LoginController	23
3.1.2	UserController	23
3.1.3	ModeratorController	23
3.1.4	AdminController	23
3.1.5	RelationController	24
3.2	Domain Model	24
3.2.1	Document	24
3.2.2	User	24
3.2.3	PublishRequest	25
3.2.4	Collection	25
3.2.5	Comment	25
3.2.6	DocumentRelation	25
3.2.7	DocumentSearchCriteria e DocumentSearchCriteriaBuilder	25
3.2.8	Tag	26
3.2.9	TagChangeRequest	26
3.3	ORM	26
3.3.1	DBConnection	26
3.3.2	BaseDAO	27
3.3.3	DocumentDAO	27
3.3.4	UserDAO	28
3.3.5	PublishRequestDAO	28
3.3.6	CollectionDAO	28
3.3.7	CommentDAO	28
3.3.8	DocumentRelationDAO	28
3.3.9	TagDAO	29
3.3.10	TagChangeRequestDAO	29

4	Testing	29
4.1	Test Business Logic	30
4.2	Test Domain Model	34
4.3	Test ORM	35

1 Introduzione

1.1 Descrizione del progetto

Questa relazione ha lo scopo di definire e spiegare i punti chiave relativi alla progettazione di un software Java ideato per la pubblicazione e manutenzione di documenti musicali. L'obiettivo principale è quello di garantire l'affidabilità e autenticità dei documenti per la corretta manutenibilità culturale. Questo, tramite moderatori che hanno conoscenza riguardo l'arte, con il ruolo di approvare i documenti pubblicati da un Autore, il cui documento si considera pubblico solo dopo tale conferma. Un utente autenticato potrà quindi visualizzare i documenti pubblici tramite una ricerca legata a vari parametri, aprire il documento, commentarlo o interagirci inserendolo in specifiche collezioni private.

1.2 Struttura del progetto e ambiente di sviluppo

Il progetto si compone in diversi package per permettere modularità e singola responsabilità. Ogni layer presenta un grado diverso di astrazione, riducendo le dipendenze e facilitando l'evoluzione del software.

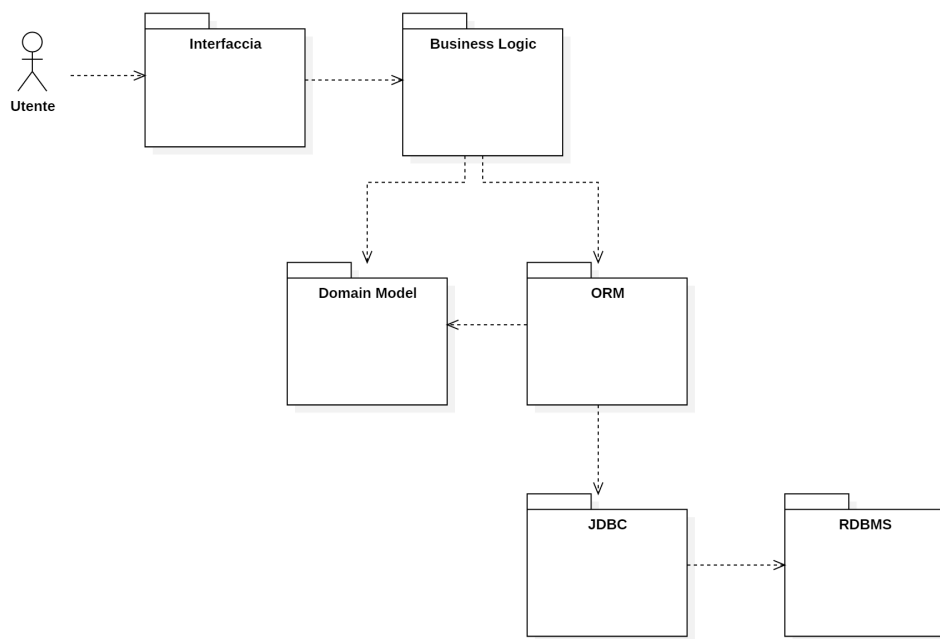


Figura 1: Architettura del progetto software

I package che troviamo nell'architettura sono i seguenti:

- **Interfaccia:** definisce come l'utente interagisce con il software, permettendo di collegare la rappresentazione grafica alle funzionalità di dominio del software. Non è stata implementata, ma una sua versione semplificata può essere compresa visualizzando i mockup presenti nel prossimo paragrafo.
- **Business Logic:** contiene la logica di business dell'architettura, definendo le funzionalità di cui ogni tipologia di utente può usufruire nel software. Il package dell'interfaccia invoca questi metodi in seguito ad un evento dell'utente.

- **Domain Model:** le classi contenute in questo package definiscono le entità del software, come i documenti (**Document**), le richieste di pubblicazione (**PublishRequest**) o i commenti (**Comment**). All'interno vivono le funzionalità di dominio, ovvero quei processi non legati alla gestione del software ma intrinseci alla natura di ogni classe, esprimendo come i dati interagiscono tra loro.
- **ORM:** questo livello contiene le classi che gestiscono la persistenza degli oggetti del dominio, trasformandoli in record del database. Qui avviene la mappatura tra classi del Domain Model e tabelle, oltre alla definizione delle operazioni di CRUD tramite l'implementazione concreta delle classi DAO.
- **JDBC:** fornisce l'accesso a basso livello al database tramite la definizioni di connessioni con esso. Tali connessioni sono fondamentali per le classi DAO, definite nell'ORM, permettendo di ottenere i risultati delle query.
- **RDBMS:** rappresenta il sistema che conserva effettivamente i dati tramite tabelle che rappresentano il dominio, vincoli e relazioni. Nel nostro caso troviamo PostgreSQL come DBMS relazionale scelto.

Parlando invece dell'ambiente di sviluppo dell'applicazione, i software utilizzati sono:

- **IntelliJ IDEA:** IDE predefinito per la scrittura del codice Java;
- **StarUML:** creazione di diagrammi UML, use case, modello ER, modello relazionale;
- **PgAdmin:** piattaforma per lo sviluppo e amministrazione di database PostgreSQL;
- **Lunacy:** software di creazione di mockup;

Il codice sorgente è reperibile nella repository GitHub seguente: https://github.com/niccobarto/swe_project.

2 Progettazione del software

2.1 Attori e casi d'uso

Il sistema prevede diverse tipologie di attori, ognuna delle quali dispone di funzionalità specifiche coerenti con il proprio ruolo nel dominio applicativo. I casi d'uso descrivono le interazioni tra gli attori e il sistema, specificando obiettivi, vincoli e dipendenze tra funzionalità.

Di seguito sono presentati i tre diagrammi dei casi d'uso, uno per ciascuna categoria di attori principali.

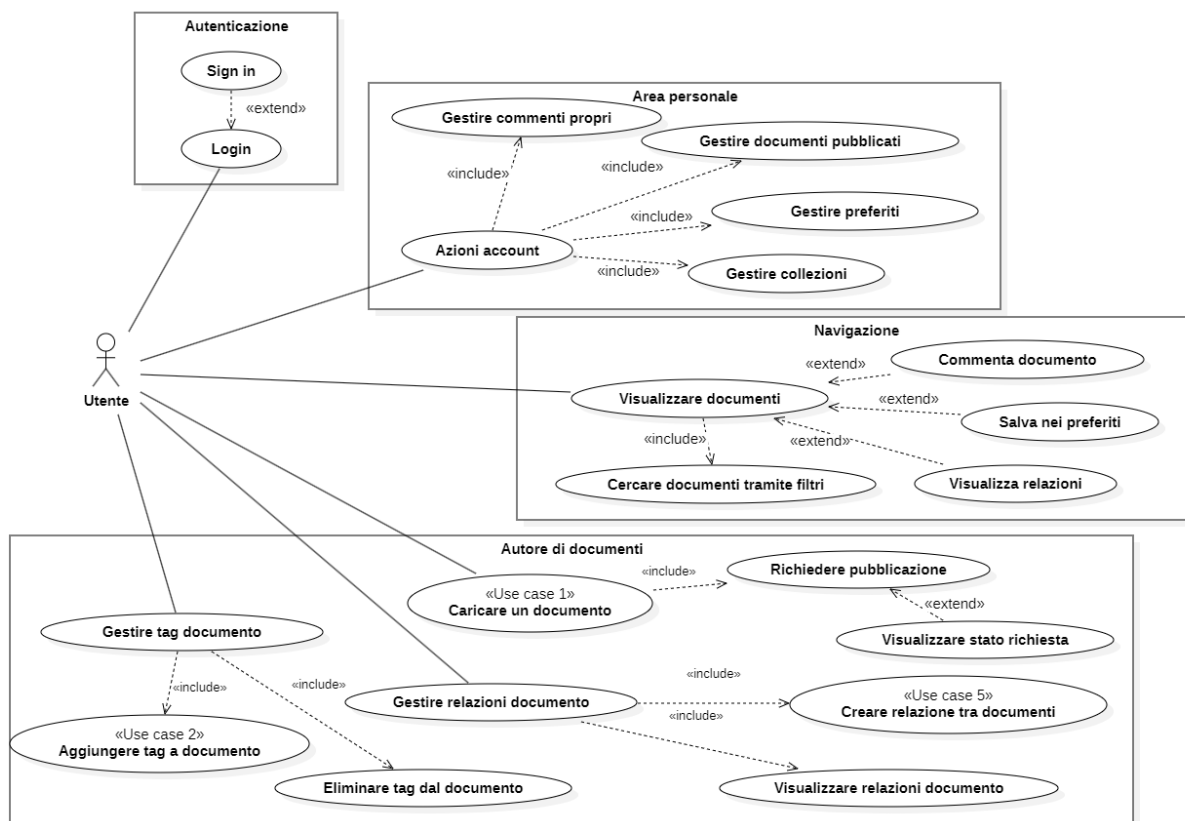


Figura 2: Casi d'uso dell'Utente / Autore

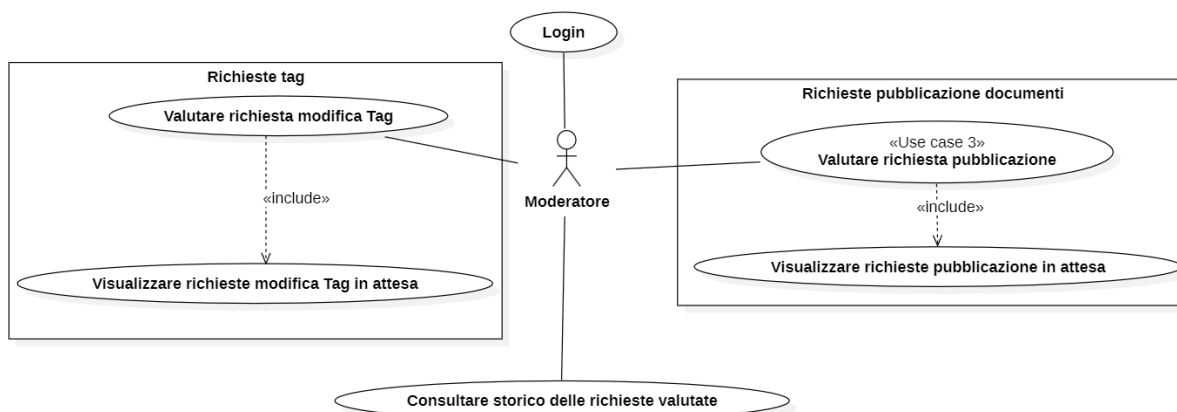


Figura 3: Casi d'uso dell'attore Moderatore

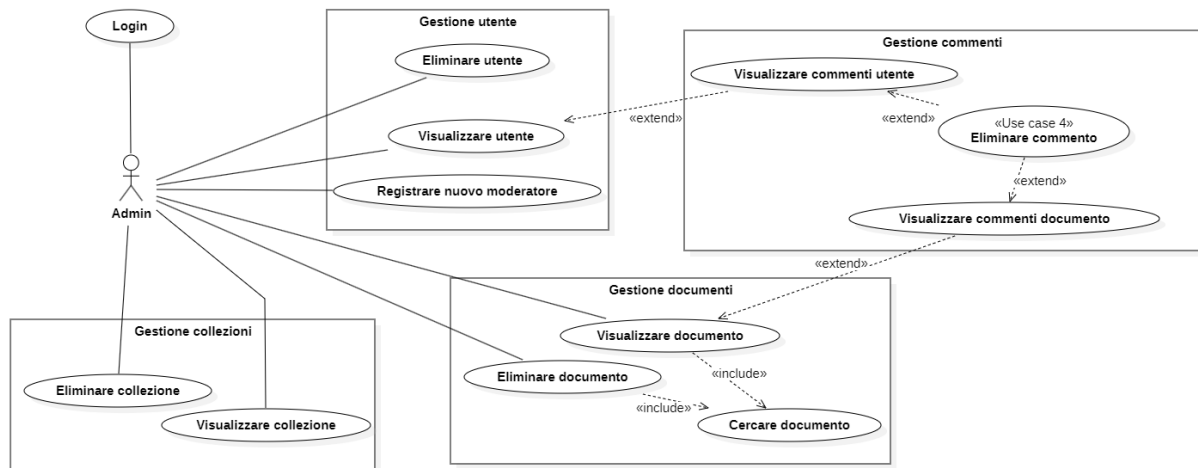


Figura 4: Casi d'uso dell'attore Admin

2.1.1 Utente predefinito

L'Utente predefinito è un soggetto autenticato che può accedere alle funzionalità del sistema adibite alla consultazione dei documenti pubblicati e la ricerca filtrata. Può rilasciare commenti sotto uno specifico documento che sarà pubblico a tutti gli altri utenti del sistema, che siano con stesso livello di accesso o moderatori/admin. Può organizzare documenti in collezioni o spuntarli come preferiti.

2.1.2 Autore

L'Autore è un utente autenticato che può creare, modificare e gestire i propri documenti. Qualsiasi utente ha la possibilità di pubblicare documenti, pertanto Autore viene considerato un tipo specifico di Utente. È la figura centrale nella produzione di contenuti: può caricare nuovi documenti, etichettarli con tag, organizzarli in collezioni, e soprattutto inviare una richiesta di pubblicazione, che verrà valutata dai Moderatori. Definisce relazioni tra documenti di propria proprietà ed altrui, ovvero collegamenti significativi a cui viene associato un significato, come "una nuova versione" oppure "trascrizione", e che unite definiscono una rete che lega documenti ad altri. Gestisce le richieste di relazione che riguardano i propri documenti pubblicati.

Use Case 1	Creazione documento
Descrizione use case	L'utente è in grado di creare documenti di un certo formato caricandone il contenuto.
Attori coinvolti	Utente (Autore)
Flusso base	<ol style="list-style-type: none"> 1. L'utente definisce tutti i parametri necessari alla creazione di un documento nella GUI. 2. Clicca il pulsante di creazione documento. 3. Il sistema genera automaticamente i campi <code>file_path</code> e <code>file_name</code>, necessari al salvataggio. 4. La creazione del documento avviene correttamente.
Flusso alternativo	<ul style="list-style-type: none"> • Se i valori passati non rispettano i tipi previsti, viene generata un'eccezione.
Post-condizioni	Ora l'utente possiede un documento associato al proprio profilo come autore. Può richiedere la pubblicazione oppure modificarlo.

Tabella 1: Template di descrizione del caso d'uso "Creazione documento"

Use Case 2	Aggiungere tag a documento
Descrizione use case	L'autore può richiedere l'aggiunta di un tag a un proprio documento. L'operazione genera sempre una richiesta di modifica dei tag, che dovrà essere valutata e approvata da un moderatore prima che il tag diventi effettivamente visibile sul documento.
Attori coinvolti	Utente (Autore), Moderatore
Flusso base	<ol style="list-style-type: none"> 1. L'autore seleziona un proprio documento. 2. Inserisce il nome del tag da aggiungere. 3. Il sistema verifica se il tag esiste già nel catalogo dei tag. 4. In entrambi i casi, il sistema crea una richiesta di modifica dei tag associata al documento. 5. La richiesta viene registrata con stato PENDING. 6. Un moderatore visualizza la richiesta e ne valuta l'approvazione.
Flusso alternativo	<ul style="list-style-type: none"> • Se esiste già una richiesta pendente per lo stesso documento e lo stesso tag, il sistema blocca la creazione di richieste duplicate. • Se l'utente non è l'autore del documento, l'operazione viene negata.
Post-condizioni	<ul style="list-style-type: none"> • In caso di approvazione, il tag viene associato al documento e reso pubblico. • In caso di rifiuto, il documento rimane invariato.

Tabella 2: Template di descrizione del caso d'uso "Aggiungere tag a documento"

Use Case 3	Creare relazioni tra documenti
Descrizione use case	Un autore può definire una relazione tra due documenti pubblicati, scegliendo il tipo di relazione (e.g. <code>NEW_VERSION_OF</code>). La relazione diviene visibile a tutti solo dopo l'accettazione da parte dell'autore del documento di destinazione.
Attori coinvolti	Utente mittente, Utente destinatario
Flusso base	<ol style="list-style-type: none"> 1. L'utente mittente seleziona un proprio documento pubblicato, che sarà il <i>source</i>. 2. Seleziona un secondo documento, che può appartenere a sé stesso oppure a un altro autore, che rappresenta il <i>destination</i>. 3. Sceglie il tipo di relazione tra <i>source</i> e <i>destination</i>. 4. Il sistema registra la relazione come "in attesa di accettazione". 5. L'autore del documento di destinazione visualizza la richiesta di relazione. 6. L'utente destinatario accetta la relazione, che diviene pubblica e accessibile a tutti gli utenti.
Flusso alternativo	<ul style="list-style-type: none"> • Se i due documenti appartengono allo stesso autore, la relazione viene pubblicata automaticamente senza alcuna richiesta. • Se l'autore del documento di destinazione non approva la relazione, essa rimane in stato non pubblicato ed eventualmente viene eliminata dal sistema. • Se uno dei documenti non è pubblicato, la relazione non può essere creata.
Post-condizioni	Una volta accettata, la relazione tra i due documenti è visibile a tutti gli utenti e favorisce la navigazione tra documenti connessi da affinità di contenuto, versione o struttura.

Tabella 3: Template di descrizione del caso d'uso "Creare relazioni tra documenti"

2.1.3 Moderatore

Il Moderatore ha il compito di garantire la qualità dei documenti pubblicati. Può visualizzare le richieste di pubblicazione in attesa, analizzare il contenuto del documento e approvarne o rifiutarne la pubblicazione. È inoltre in grado di consultare lo storico delle richieste già valutate, garantendo tracciabilità delle decisioni. La sua gestione si estende all'assegnazione di un Tag ad un documento, garantendo che gli Autori non associno label

non conformi al tipo di documento o al suo contenuto.

Use Case 4	Gestire richiesta pubblicazione documento
Descrizione use case	Il moderatore analizza una richiesta di pubblicazione relativa a un documento caricato da un autore, visualizzandone contenuto e metadati, e decide se approvarla o rifiutarla.
Attori coinvolti	Moderatore
Flusso base	<ol style="list-style-type: none"> 1. Il moderatore accede alla lista delle richieste di pubblicazione in attesa. 2. Seleziona una richiesta e visualizza il documento, i suoi metadati e le informazioni fornite dall'autore. 3. Valuta il contenuto e decide se approvare o rifiutare la richiesta. 4. In caso di approvazione, il sistema imposta lo stato del documento a APPROVED e lo rende pubblico. 5. La richiesta viene marcata come chiusa.
Flusso alternativo	<ul style="list-style-type: none"> • In caso di rifiuto, il sistema imposta lo stato a REJECTED e permette al moderatore di inserire una motivazione. • Se l'utente che tenta la valutazione non è un moderatore (o admin), l'operazione viene bloccata. • Il documento deve trovarsi in stato DRAFT prima della valutazione: in caso contrario la richiesta è invalida.
Post-condizioni	<p>A seconda dell'esito:</p> <ul style="list-style-type: none"> • Se APPROVED, il documento diventa pubblico e consultabile da tutti gli utenti autenticati. • Se REJECTED, il documento rimane privato e l'autore può modificarlo e inviare una nuova richiesta.

Tabella 4: Template di descrizione del caso d'uso "Gestire richiesta pubblicazione"

2.1.4 Admin

L'Admin è il super-utente del sistema, con accesso a tutte le funzionalità gestionali. Può amministrare utenti, documenti, collezioni e commenti, svolgendo funzioni di supervisione e mantenimento dell'integrità del sistema. Sceglie tra gli Utenti chi diventa Moderatore e ne affida le funzionalità di giudice di documenti citati precedentemente. Gli use case

dell'Admin includono sia funzionalità correttive (ad esempio eliminazione contenuti) sia funzionalità amministrative (gestione ruoli).

Use Case 5	Eliminazione di un commento
Descrizione use case	L'amministratore può rimuovere un commento pubblicato da un qualsiasi utente qualora risulti inappropriato, offensivo o non conforme alle linee guida della piattaforma.
Attori coinvolti	Admin
Flusso base	<ol style="list-style-type: none"> 1. L'admin accede alla lista dei commenti associati ai documenti oppure associati ad uno specifico utente. 2. Seleziona il commento da eliminare. 3. Conferma l'operazione di eliminazione. 4. Il sistema rimuove definitivamente il commento dal database e aggiorna l'interfaccia utente.
Flusso alternativo	<ul style="list-style-type: none"> • Se il commento non esiste più (già eliminato o riferito a un documento rimosso), il sistema segnala l'errore. • Se l'utente che effettua l'operazione non ha privilegi da admin, l'azione viene bloccata.
Post-condizioni	Il commento selezionato viene eliminato dal sistema, migliorando la qualità dei contenuti pubblici e mantenendo conforme lo spazio di discussione.

Tabella 5: Template di descrizione del caso d'uso "Eliminazione di un commento"

2.2 Mockups

Poiché l'interfaccia grafica non è stata implementata direttamente nel progetto, sono stati realizzati dei mockup ad alta fedeltà con lo scopo di rappresentare in modo chiaro le principali interazioni tra utente e sistema.

I mockup permettono di visualizzare il flusso dei casi d'uso descritti nei paragrafi precedenti, mostrando come le funzionalità di creazione documenti, gestione dei tag, moderazione dei contenuti e amministrazione degli utenti siano esposte all'utente finale in modo intuitivo e coerente.

In particolare:

- la creazione di un documento evidenzia la distinzione tra salvataggio in bozza e richiesta di pubblicazione;
- l'aggiunta dei tag segue un workflow controllato, basato su richieste di approvazione;
- il moderatore dispone di pannelli dedicati per la gestione delle richieste pendenti;

- l'amministratore può intervenire sugli utenti e sui commenti per garantire il corretto utilizzo della piattaforma.

The mockup shows the Scriptoria application interface. At the top, there is a header with the Scriptoria logo and the text 'User: Nome'. Below the header is a navigation bar with links: 'My Documents', 'Search', 'Collections', and 'Favourites'. The 'My Documents' link is highlighted. The main content area displays a modal window titled 'Add New Document' with a close button (X) in the top right corner. The form contains the following fields and options:

- Title:** A text input field with placeholder text 'Enter the document title'.
- Description:** A text input field with placeholder text 'Enter a brief description of the document'.
- Period:** A text input field with placeholder text 'Enter period'.
- Format:** Radio buttons for PDF (selected), MIDI, Audio, XML, TXT, and Image.
- Tags (optional):** A text input field with placeholder text 'Add a tag'. Below it, a tag 'Baroque' is shown with a close button (X).
- File Upload:** A button labeled 'Select file' with a file icon.

At the bottom right of the form are two buttons: 'Submit' and 'Submit & Request Publication', both with checkmark icons.

Figura 5: Mockup: creazione di un nuovo documento

The mockup shows the Scriptoria application interface. At the top, there is a header with the Scriptoria logo and the text 'User: Nome'. Below the header is a navigation bar with links: 'My Documents', 'Search', 'Collections', and 'Favourites'. The 'My Documents' link is highlighted. The main content area displays a modal window titled 'Add Tag' with a close button (X) in the top right corner. The document title 'Mahler Symphony n1' is displayed at the top of the modal. Below the title, there is a search bar with the text 'Search for an existing tag or propose new one.' and a search icon. The search bar contains the text 'Romanticism'. Below the search bar, a message states 'NO RESULT FOUND, PLEASE REQUEST TO ADD THE NEW TAG.' At the bottom right of the modal is a button labeled 'Request Add Tag' with a checkmark icon.

Figura 6: Mockup: aggiunta di un tag a un documento

Scriptoria User: Nome

My Documents Search Collections Favourites

Create Document Relationship

From	Relationship type	To
Select source document	<input checked="" type="radio"/> Quote <input type="radio"/> Transcribes <input type="radio"/> Answer to <input type="radio"/> New version of <input type="radio"/> Other	Select destination document

(a) Creazione di una relazione tra documenti lato sorgente

Scriptoria User: Nome

My Documents Search Collections Favourites


Mario Rossi has requested to relate one of your document

Mario Rossi has requested to relate: "Mahler Symphony No.2"
 new version of your document : "Mahler Symphony No2. , Resurrection"

Mahler Symphony No2
 New version of
 Mahler Symphony No2, Resurrection

(b) Creazione di una relazione tra documenti lato destinatario

Figura 7: Mockup: Creazione di una relazione tra due documenti appartenenti a utenti autori differenti


Scriptoria
















Moderator: Nome

Pending Requests



Request History

Publish Requests


Tag Requests

Document	User	Requested on	Actions	
Ewald brass quintet n1 PDF	Mario Rossi	Apr 14, 2025	View details 	 
Mahler symphony n3 XML	Chiara Verdi	May 25, 2025	View details 	 
Beethoven piano concerto TXT	Francesca Marri	Aug 15, 2025	View details 	 
Avengers Soundtrack MIDI	Giovanni Parri	Aug 27, 2025	View details 	 
Ewald brass quintet n2 PDF	Mario Rossi	Dec 1, 2025	View details 	 

1/3

(a) Richieste di pubblicazione


Scriptoria











Moderator: Nome

Pending Requests



Request History

Publish Requests

Tag Requests

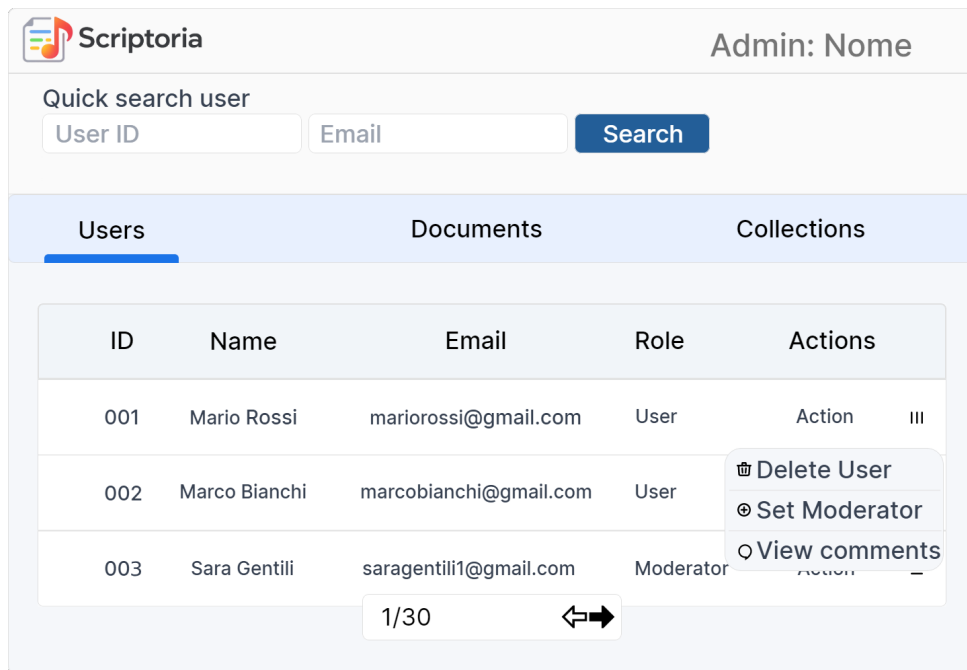
Document	User	Requested on	Tag	Actions
Ewald brass quintet n1 PDF	Mario Rossi	Apr 20, 2025	Baroque	 
Mahler symphony n3 XML	Chiara Verdi	June 6, 2025	Brass	 
Beethoven piano concerto TXT	Francesca Marri	Aug 26 2025	Symphonic	 
Avengers Soundtrack MIDI	Giovanni Parri	Sep 9, 2025	Quartet	 
Ewald brass quintet n2 PDF	Mario Rossi	Dec 14, 2025	Band	 

1/6

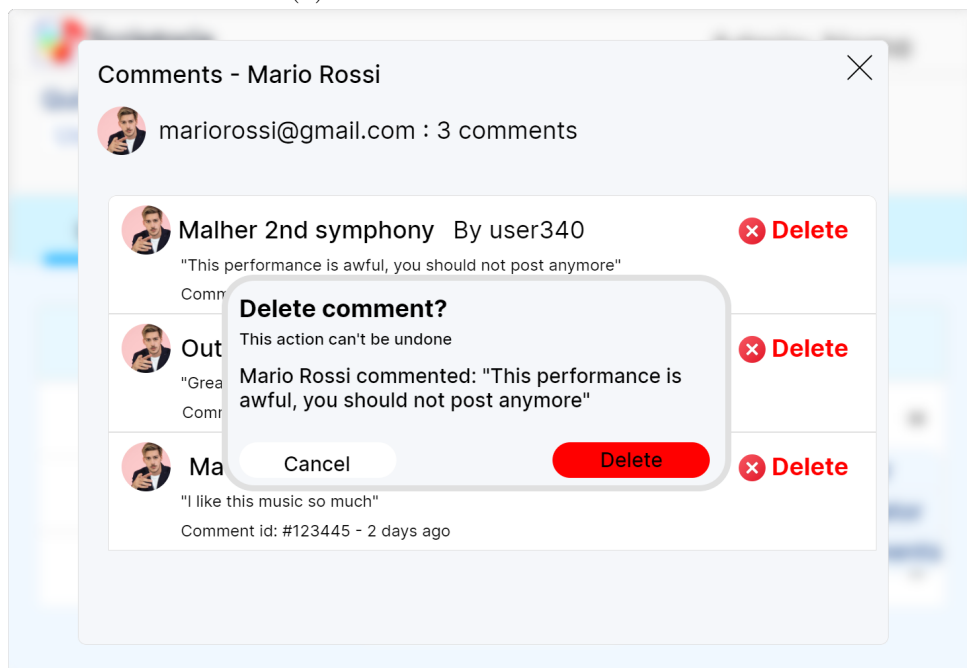



(b) Richieste di modifica dei tag

Figura 8: Mockup: pannello di moderazione delle richieste



(a) Pannello di amministrazione



(b) Eliminazione di un commento

Figura 9: Mockup: funzionalità amministrative

2.3 Class diagram

Il diagramma delle classi fornisce una visuale dell'architettura del sistema, descrivendo le principali classi coinvolte, i loro attributi, le funzioni che forniscono e le relazioni che intercorrono tra di esse. Rispetto alla sola suddivisione in package presentata in precedenza, il class diagram consente di evidenziare in modo più preciso le responsabilità di ciascuna classe.

Le classi sono organizzate nei tre package principali:

- **Domain Model:** contiene le entità del dominio applicativo, cioè i concetti con cui si interagisce nell'utilizzo dell'applicazione (ad esempio User, Document, Collection, PublishRequest, Comment, DocumentSearchCriteria, DocumentSearchCriteriaBuilder) e le relative enumerazioni (DocumentStatus, DocumentFormat, RequestStatus, DocumentRelationType);

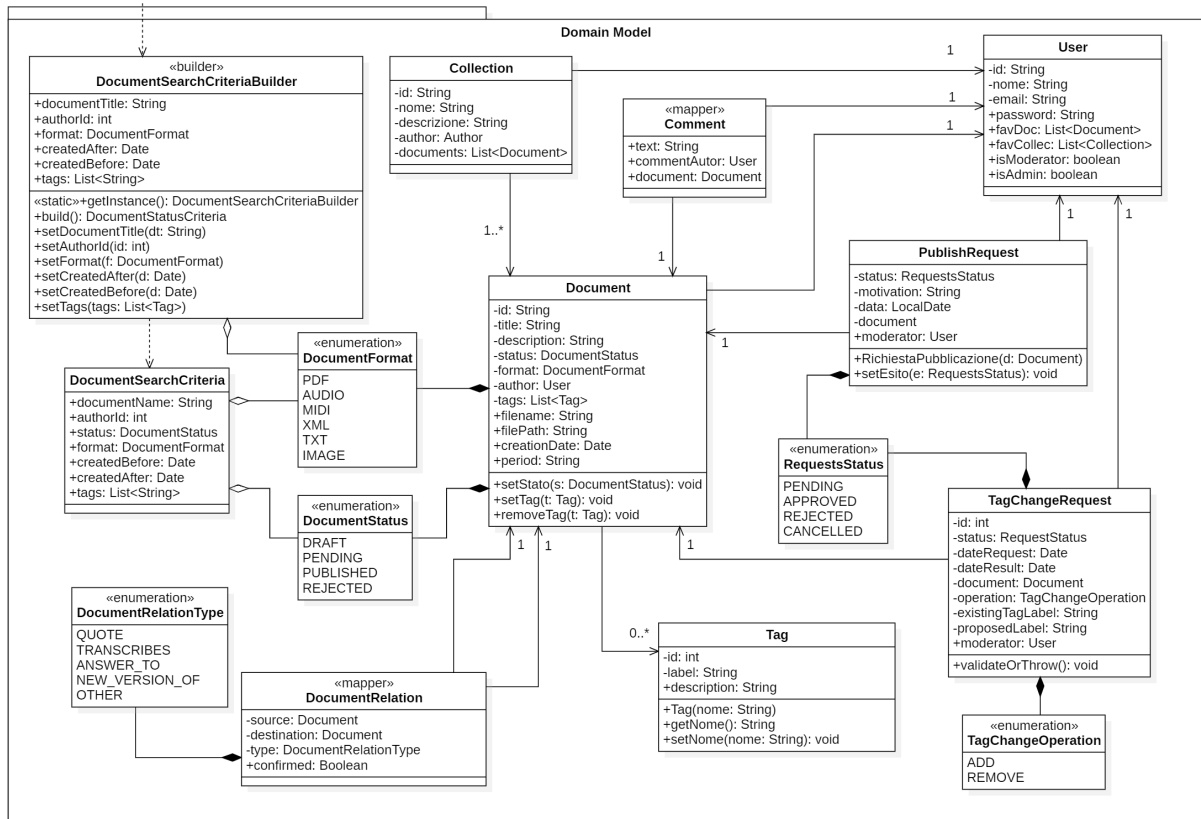


Figura 10: Domain Model

- **Business Logic:** comprende i controller responsabili dell'implementazione dei casi d'uso (ad esempio UserController, AdminController, ModeratorController, RelationController, LoginController) e che orchestrano le operazioni sulle entità del dominio;

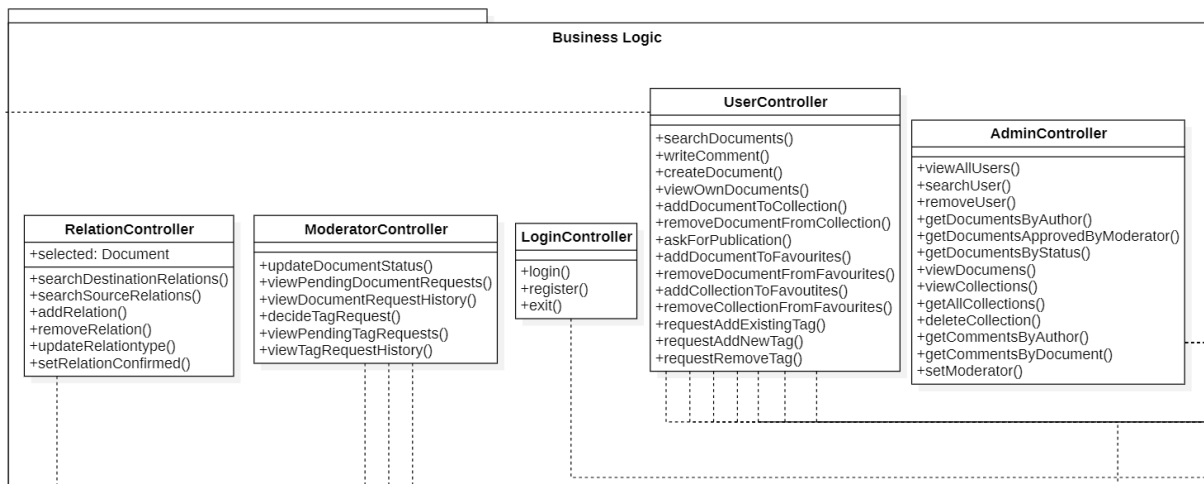


Figura 11: Business Logic

- **ORM (Object Relation Mapping)**: raccoglie le classi DAO (UserDAO, DocumentDAO, PublishRequestDAO, CommentDAO, CollectionDAO, DocumentsRelationDAO, TagDAO, TagChangeRequestDAO) e le componenti di accesso al database (BaseDAO, DBConnection), incaricate di tradurre le operazioni sul modello a oggetti in query verso il database relazionale.

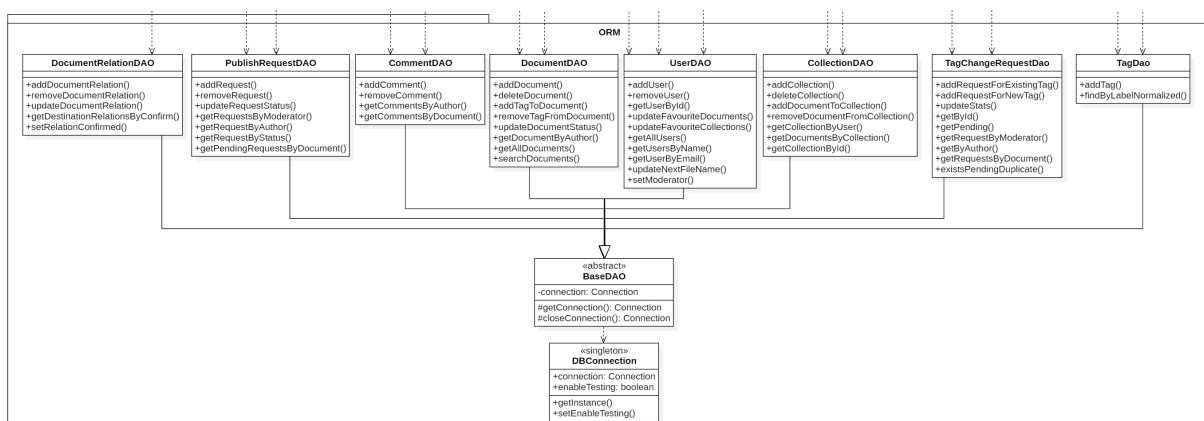


Figura 12: ORM (Object Relation Mapping)

2.3.1 Design pattern: Singleton

Considerando che ogni classe DAO presente nel package ORM per funzionare deve ottenere una connessione al database, si vuole definire un meccanismo che permetta di centralizzare questo processo. La classe **DBConnection** implementa il pattern creazionale **Singleton**, utilizzato per la garanzia dell'esistenza di un'unica istanza di connessione al database.

Il costruttore della classe è dichiarato **private**, impedendo la creazione di oggetti tramite il costruttore. L'accesso alla connessione avviene esclusivamente attraverso il metodo **getConnection()** della classe astratta **BaseDAO**, estesa da tutte le classi DAO, che istanzia la connessione al database chiamando il metodo statico **getInstance()** di **DBConnection**. Tale metodo richiama il costruttore privato solamente al primo utilizzo, ritornando l'istanza appena creata oppure la cui creazione era avvenuta in invocazioni precedenti.

2.3.2 Design pattern: Mapper

Prendendo la classe **Comment** nel **Domain Model**, non rappresenta di per se un'entità autonoma, ma collegata direttamente al documento a cui è riferita e all'utente autore del commento. Il suo ruolo principale, quindi, è quello di **mappare** la relazione tra **User** e **Document**, aggiungendo come informazione il testo di tale commento.

Tale design pattern viene chiamato **Mapper**. Anche alla classe **DocumentRelation** associamo le stesse caratteristiche, collegando due istanze di **Document** e includendo il tipo di relazione che c'è nella coppia.

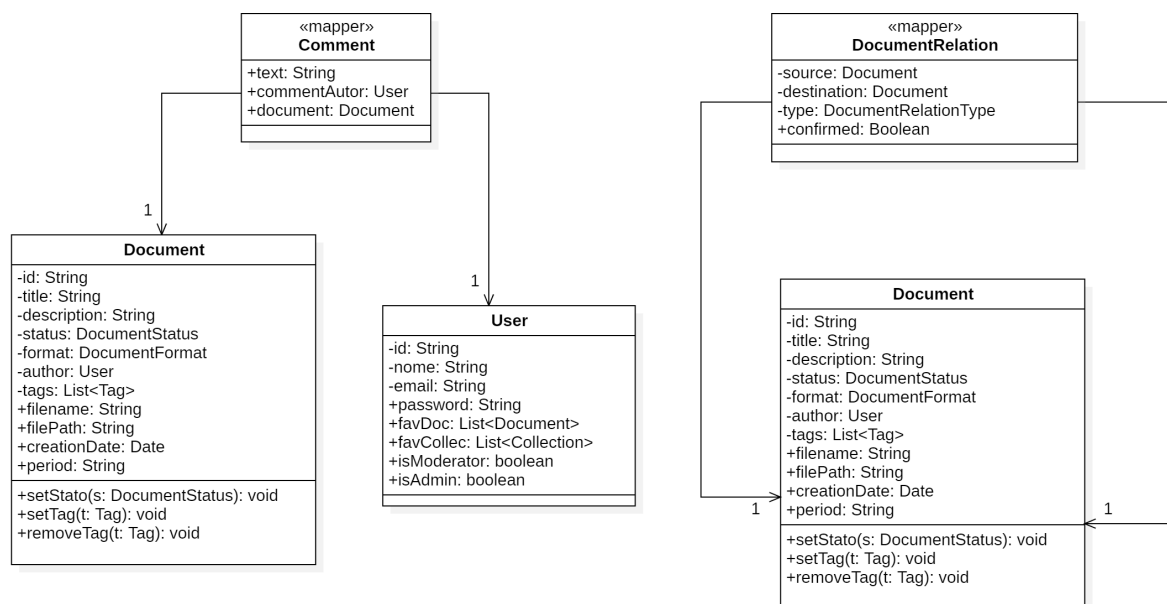


Figura 13: Mapper

2.3.3 Design pattern: Builder

La classe **DocumentSearchCriteria** è stata introdotta per rappresentare in modo compatto i possibili filtri di ricerca sui documenti, come il titolo, l'autore, lo stato del documento, il formato, la data di creazione e l'insieme di tag associati. Poiché è presente un vasto spazio dei parametri possibili e non tutti potrebbero essere utilizzati nella stessa ricerca, è necessario definire un meccanismo che ci distanzi dall'implementazione di numerosi costruttori per considerare ogni possibile gruppo di parametri, un problema denominato *telescoping constructor*.

Per evitare questo problema è stato adottato il pattern *Builder*. La classe **DocumentSearchCriteriaBuilder** funge da costruttore dedicato: essa espone metodi che permettono di impostare solo i campi di interesse e, tramite il metodo **build()**, restituisce un'istanza di **DocumentSearchCriteria**. In questo modo è possibile creare criteri di ricerca complessi senza ricorrere a costruttori con lunghi elenchi di parametri, migliorando la leggibilità del codice e rispettando l'*Open/Closed Principle*. Infatti l'aggiunta di un nuovo parametro di ricerca non richiederà la modifica del codice qui mostrato, ma soltanto l'estensione con l'aggiunta del parametro desiderato e del relativo metodo **set...()**

Nella pratica, la Business Logic costruisce un oggetto **DocumentSearchCriteria** tramite il builder a partire dai filtri selezionati dall'utente, e lo passa ai DAO responsabili della generazione della query SQL.

```

1 public class DocumentSearchCriteriaBuilder {
2     private String documentTitle;
3     private Integer authorId;
4     private DocumentFormat format;
5     private Date createdAfter;
6     private Date createdBefore;
7     private List<String> tags;
8
9     public DocumentSearchCriteriaBuilder setDocumentTitle(String
documentTitle) {
10         this.documentTitle = documentTitle;
11         return this;
12     }
13     public DocumentSearchCriteriaBuilder setAuthorId(Integer authorId) {
14         this.authorId = authorId;
15         return this;
16     }
17
18     public DocumentSearchCriteriaBuilder setFormat(DocumentFormat format
) {
19         this.format = format;
20         return this;
21     }
22     public DocumentSearchCriteriaBuilder setCreatedAfter(Date
createdAfter) {
23         this.createdAfter = createdAfter;
24         return this;
25     }
26     public DocumentSearchCriteriaBuilder setCreatedBefore(Date
createdBefore) {
27         this.createdBefore = createdBefore;
28         return this;
29     }
30     public DocumentSearchCriteriaBuilder setTags(List<String> tags) {
31         this.tags = tags;
32         return this;
33     }
34
35     private DocumentSearchCriteriaBuilder(){}
36
37     public static DocumentSearchCriteriaBuilder getInstance(){
38         return new DocumentSearchCriteriaBuilder();
39     }
40
41     public DocumentSearchCriteria build() {
42         DocumentSearchCriteria criteria = new DocumentSearchCriteria();
43         criteria.setDocumentTitle(this.documentTitle);
44         criteria.setAuthorId(this.authorId);
45         criteria.setFormat(this.format);
46         criteria.setCreatedAfter(this.createdAfter);
47         criteria.setCreatedBefore(this.createdBefore);
48         criteria.setTags(this.tags);
49         return criteria;
50     }

```

Listing 1: Codice della classe DocumentSearchCriteriaBuilder

2.3.4 Design pattern: DAO

L'obiettivo del pattern *Data Access Object* (DAO) è separare la logica di accesso ai dati dalla logica di business, incapsulando tutte le operazioni verso il database in classi dedicate. In questo modo il codice applicativo non dipende da dettagli specifici di SQL, del driver JDBC o della struttura fisica del database, ma interagisce con un'interfaccia ad oggetti dedicata all'interazione con esso.

Nel progetto ogni entità principale del dominio ha un proprio DAO dedicato, responsabile delle operazioni di creazione, lettura, aggiornamento e cancellazione (CRUD). Ad esempio: `UserDAO` si occupa della persistenza degli oggetti `User`, `DocumentDAO` gestisce gli oggetti `Document`, `PublishRequestDAO` le entità `PublishRequest`, `CommentDAO` i commenti. Tutte queste classi sono raccolte nel package `ORM`, che funge da strato di accesso ai dati dell'architettura.

2.4 Diagramma ER e Modello Relazionale

Il diagramma ER (Entità-Relazione) definisce la rappresentazione concettuale dei dati, costruendo entità, con i relativi attributi, e relazioni che associano due entità seguendo un significato semantico. In generale, la definizione di un modello concettuale permette di concentrarsi inizialmente sulla rappresentazione dei dati provenienti da un contesto reale, senza doverli strutturare in un DB.

Il passaggio al Modello Relazionale determina la fase di traduzione dello schema concettuale, dove le entità diventano relazioni effettive (tabelle), mentre gli attributi costituiscono le colonne di tali tabelle. Le relazioni del Modello ER indicano dei vincoli tra le tabelle del Modello Relazionale, permettendo di identificare vincoli di integrità referenziale. Talvolta, i legami tra due entità del modello concettuale sono abbastanza complessi da dover definire una relazione per rappresentarla nel Modello Relazionale (è di esempio la relazione `Comment`).

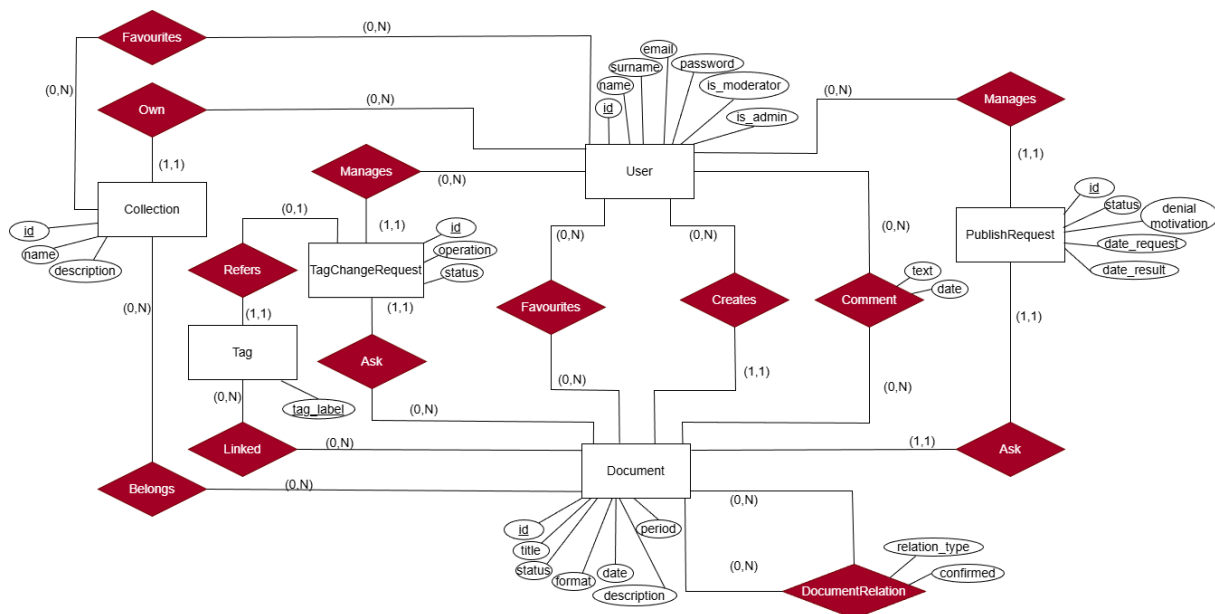


Figura 14: Diagramma ER

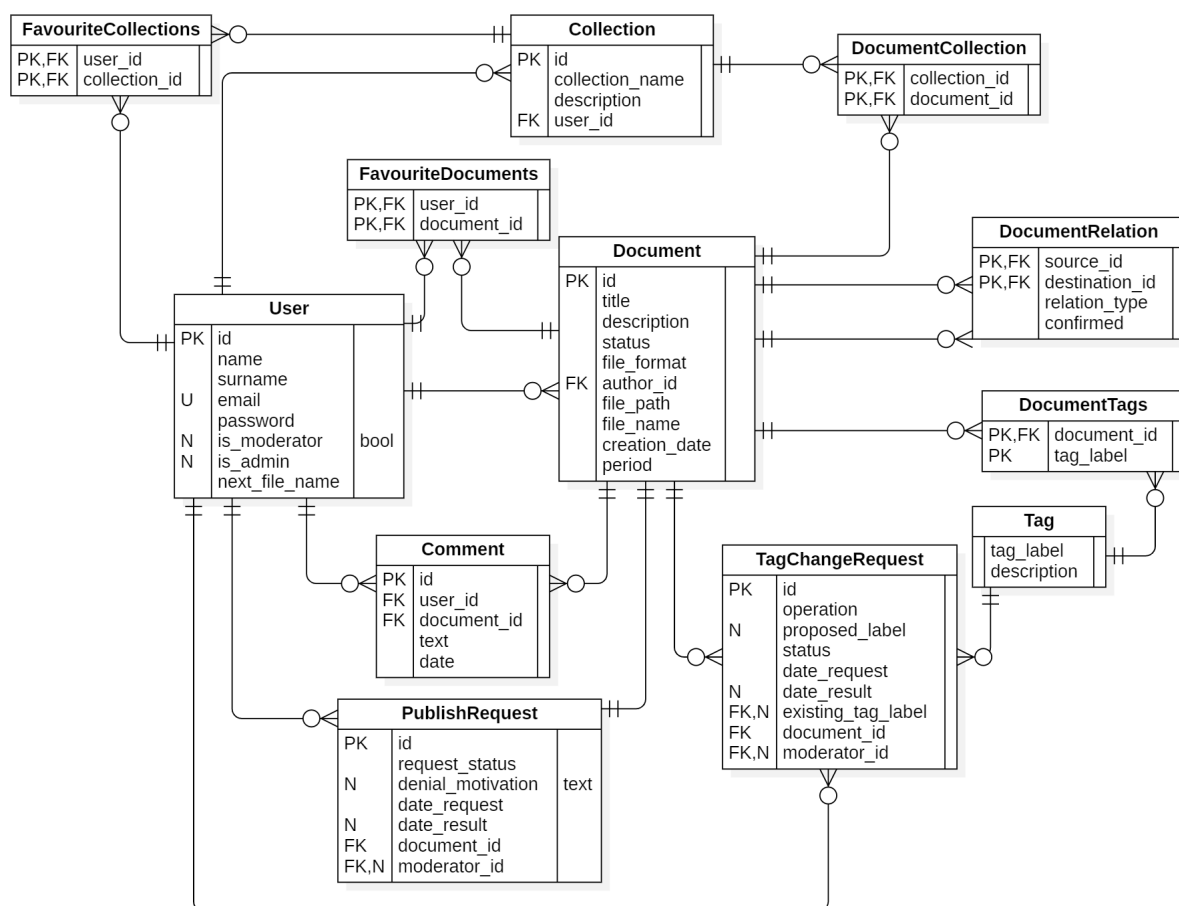


Figura 15: Modello relazionale

3 Implementazione delle classi

3.1 Business Logic

Come definito nelle sezioni precedenti, la Business Logic rappresenta il package che espone tutte le funzionalità utilizzabili dagli utenti del sistema. Questo livello fornisce un insieme di controller che mettono a disposizione dell'interfaccia utente i metodi necessari alla realizzazione dei casi d'uso, orchestrando le classi del Domain Model e delegando le operazioni di persistenza al package ORM.

3.1.1 LoginController

Il `LoginController` rappresenta il primo punto di contatto tra l'utente e il sistema, poiché l'accesso alle funzionalità dell'applicazione è subordinato a un processo di autenticazione. Esso espone le operazioni di `login` e `register`, consentendo all'utente di autenticarsi o di creare un nuovo account. Una volta completata con successo l'autenticazione, il flusso applicativo viene demandato allo `UserController`.

3.1.2 UserController

Lo `UserController` espone le funzionalità di base accessibili a qualsiasi utente autenticato. Tra queste rientrano operazioni come `createDocument`, `searchDocuments` e `writeComment`. Attraverso questo controller l'utente può creare e gestire collezioni personali, aggiungere documenti alle collezioni, scrivere commenti sui documenti pubblici e ricercare documenti tramite filtri. Inoltre, qualora l'utente assuma anche il ruolo di autore, lo `UserController` consente di accedere alle funzionalità di creazione e modifica dei documenti, inclusa l'aggiunta di tag e la definizione di relazioni con altri documenti.

3.1.3 ModeratorController

Il `ModeratorController` raccoglie le funzionalità riservate agli utenti con ruolo di moderatore. In questo contesto il moderatore può selezionare una richiesta, che può riguardare la pubblicazione di un documento o la modifica di un tag dello stesso, visualizzandone i dettagli e determinarne l'esito, approvandola o rifiutandola. L'accesso a tali operazioni è protetto da controlli sui permessi, poiché le funzionalità sono disponibili esclusivamente agli utenti che hanno ottenuto l'abilitazione da parte di un amministratore.

3.1.4 AdminController

L'`AdminController` fornisce le funzionalità amministrative e di supervisione del sistema, accessibili solo agli utenti con privilegio `admin`. Centralizza il controllo dei permessi tramite `ensureAdmin()`, e usa i DAO per operazioni di consultazione e gestione su utenti, documenti, collezioni e commenti (es. rimozione commenti, eliminazione collezioni, assegnazione ruolo moderatore). Include anche funzioni di report/consultazione come l'elenco documenti per autore o per stato e la consultazione dei documenti approvati da uno specifico moderatore.

3.1.5 RelationController

La scelta di includere un controller dedicato ad una specifica funzionalità è definita dalla sua importanza e complessità, staccandolo quindi dai metodi di UserController. L'attributo `selected` presente nel controller rappresenta il documento selezionato dall'utente per eseguire la prossima azione, che può essere:

- ricercare quali relazioni hanno `selected` come source, specificando eventualmente il tipo di relazione che si cerca;
- ricercare quali relazioni hanno `selected` come destination, specificando eventualmente il tipo di relazione che si cerca;
- aggiungere una relazione che veda `selected` come source;
- eliminare una relazione che veda `selected` come source;
- accettare la richiesta di relazione effettuata da un'altro utente, dove `selected` è la destination della relazione;

3.2 Domain Model

Il Domain Model rappresenta il nucleo concettuale dell'applicazione e descrive le entità fondamentali del dominio, le loro proprietà e le relazioni che intercorrono tra di esse. Le classi di questo package modellano i concetti chiave del sistema di gestione dei documenti musicali e sono utilizzate dalla Business Logic per implementare i casi d'uso, mentre la loro persistenza è dipendente dal package ORM.

3.2.1 Document

La classe `Document` rappresenta l'entità centrale del dominio applicativo. Un documento modella un contenuto musicale caricato da un autore ed è caratterizzato da attributi descrittivi quali il titolo, la descrizione, il formato del file (`DocumentFormat`), il periodo storico di riferimento e la data di creazione. Ogni documento è associato a uno stato (`DocumentStatus`) che ne definisce il ciclo di vita, ad esempio bozza, in attesa di approvazione o pubblicato.

Un documento è sempre associato a un autore, rappresentato da un'istanza della classe `User`, ed è inoltre collegabile ad altri documenti tramite relazioni.

3.2.2 User

La classe `User` rappresenta un utente del sistema. Essa contiene le informazioni necessarie all'identificazione e all'autenticazione, come nome, cognome, email e password, oltre a flag che indicano il ruolo dell'utente all'interno della piattaforma (`is_moderator`, `is_admin`).

Un utente può assumere ruoli differenti: utente standard, autore, moderatore o admin. Dal punto di vista del dominio, un utente può creare documenti, scrivere commenti, gestire collezioni personali e interagire con i contenuti pubblicati.

3.2.3 PublishRequest

La classe `PublishRequest` modella il processo di richiesta di pubblicazione di un documento. Essa rappresenta una richiesta inviata da un autore per rendere pubblico un documento precedentemente creato. Ogni richiesta è associata a un singolo documento e può essere gestita da un moderatore.

La richiesta possiede uno stato (`RequestStatus`) che ne rappresenta l'esito (`PENDING`, `APPROVED`, `REJECTED`), oltre a informazioni temporali relative alla data di invio e di valutazione. In caso di rifiuto, è possibile associare una motivazione testuale. Questa classe consente di mantenere traccia delle decisioni di moderazione e di separare chiaramente il concetto di documento da quello di pubblicazione.

3.2.4 Collection

La classe `Collection` rappresenta un insieme di documenti creato da un utente per organizzare e consultare i contenuti in modo strutturato. Ogni collezione è associata a un singolo utente proprietario, ma può contenere un numero arbitrario di documenti.

Dal punto di vista del dominio, le collezioni permettono agli utenti di classificare i documenti secondo criteri personali, indipendentemente dallo stato di pubblicazione degli stessi.

3.2.5 Comment

La classe `Comment` modella un commento testuale associato a un documento. Essa funge da mapper tra `User` e `Document`, collegando l'utente autore del commento al documento a cui il commento si riferisce. Oltre ai riferimenti alle entità coinvolte, il commento contiene il testo inserito dall'utente e la data di creazione.

I commenti permettono l'interazione tra gli utenti e arricchiscono i documenti pubblicati con osservazioni, note o discussioni. La separazione del commento come entità autonoma consente una gestione flessibile e una possibile moderazione dei contenuti.

3.2.6 DocumentRelation

La classe `DocumentRelation` rappresenta una relazione semantica tra due documenti. Ogni relazione collega una coppia di documenti distinguendo un documento sorgente (*source*) e uno di destinazione (*destination*), ed è caratterizzata da un tipo di relazione (`DocumentRelationType`) e da uno stato di conferma.

Un documento può essere in relazione con molti altri documenti, ma per ogni coppia specifica può esistere al massimo una relazione. Questa scelta progettuale consente di evitare duplicazioni e di mantenere una rappresentazione chiara dei collegamenti tra documenti, come ad esempio versioni successive, traduzioni o relazioni tematiche.

3.2.7 DocumentSearchCriteria e DocumentSearchCriteriaBuilder

Le classi `DocumentSearchCriteria` e `DocumentSearchCriteriaBuilder` sono utilizzate per modellare i criteri di ricerca dei documenti. `DocumentSearchCriteria` rappresenta un oggetto immutabile che raccoglie i possibili filtri applicabili a una ricerca, come titolo, autore, formato o data di creazione.

La costruzione di tale oggetto avviene tramite il pattern *Builder*, implementato dalla classe `DocumentSearchCriteriaBuilder`. Questo approccio evita il problema dei cosiddetti *telescoping constructors* e permette di impostare in modo chiaro e leggibile solo i criteri desiderati. La Business Logic utilizza il builder per creare i criteri di ricerca a partire dall'input dell'utente e li passa ai DAO, che si occupano di tradurli nelle opportune query verso il database.

3.2.8 Tag

La classe `Tag` rappresenta un'etichetta riutilizzabile, utilizzata per categorizzare i documenti. I tag consentono una classificazione trasversale dei contenuti e supportano la ricerca filtrata.

3.2.9 TagChangeRequest

La classe `TagChangeRequest` rappresenta una richiesta di modifica dei tag associati a un documento. Essa consente a un autore di proporre l'aggiunta o la rimozione di un tag, demandando la decisione finale a un moderatore del sistema.

La richiesta può riferirsi a un tag già esistente, identificato tramite `existingTagLabel`, oppure proporre la creazione di un nuovo tag mediante `proposedLabel`. Le due modalità sono mutuamente esclusive: i nuovi tag vengono creati solo in seguito all'approvazione della richiesta, evitando la presenza di tag non associati ad alcun documento.

La classe fornisce factory method dedicati per la creazione delle due tipologie di richiesta e mantiene direttamente nel Domain Model le principali invarianti di correttezza, verificate dal metodo `validateOrThrow()`.

3.3 ORM

Il package ORM rappresenta il livello di separazione tra i dati dell'applicazione e il suo dominio. Il suo compito è quello di tradurre le operazioni eseguite sugli oggetti del Domain Model in interrogazioni verso il database relazionale e viceversa. Le classi DAO contenute nel package rispondono alle interrogazioni istanziando classi del Domain Model, oppure aggiornando il database in base alle operazioni eseguite dall'utente. Le classi DAO citate precedentemente incapsulano l'utilizzo di JDBC.

3.3.1 DBConnection

La classe `DBConnection` si occupa della gestione della connessione al database ed è implementata secondo il pattern Singleton. Tale scelta garantisce che l'intera applicazione utilizzi un unico punto di accesso alla connessione, evitando la creazione incontrollata di connessioni multiple.

La connessione viene inizializzata solo al primo utilizzo (lazy initialization) e resa disponibile ai DAO tramite un metodo statico.

La classe implementa un attributo `enableTesting` che, se `True` all'invocazione del metodo `getInstance()`, restituisce la connessione ad un database parallelo a quello di produzione, adibito a testare le funzionalità del software. In questo modo priviamo il database di produzione di operazioni differenti rispetto al suo utilizzo verso l'utenza.

3.3.2 BaseDAO

La classe `BaseDAO` rappresenta una superclasse astratta comune a tutti i DAO del progetto. Essa fornisce funzionalità condivise, come l'accesso alla connessione ottenuta tramite `DBConnection`, e definisce una base comune per la gestione delle operazioni. Contiene l'attributo `Connection connection`, il cui tipo è contenuto nella libreria `java.sql` e che viene restituito dal metodo `getConnection()` ereditato da tutte le classi che estendono `BaseDAO`.

3.3.3 DocumentDAO

Il `DocumentDAO` è responsabile della persistenza degli oggetti `Document`. Espone metodi per la creazione, l'aggiornamento, l'eliminazione e il recupero dei documenti dal database.

Un aspetto rilevante di questo DAO è la gestione delle operazioni di ricerca: il `DocumentDAO` riceve un'istanza di `DocumentSearchCriteria` e costruisce dinamicamente la query SQL applicando esclusivamente i filtri specificati. In questo modo, la logica di costruzione delle query rimane confinata nel livello ORM, mentre la Business Logic si limita a definire i criteri di ricerca desiderati.

Nel codice presente in Listing 2 viene invece mostrato un esempio di interazione tra ORM, database e Domain Model per il metodo di creazione di un documento .

```
1 public boolean addDocument(User author,
2                             String title,
3                             String description,
4                             String documentPeriod,
5                             DocumentFormat documentFormat,
6                             String filePath,
7                             String fileName,
8                             List<String> tags){
9
10     try{
11         String query = "INSERT INTO document (file_name,description,
12         status,period,file_format,file_path,author_id,creation_date,title)
13         VALUES(?,?,?,?,?,?,?,?,?)";
14
15         PreparedStatement statement = connection.prepareStatement(
16         query);
17
18         statement.setString(1, fileName);
19         statement.setString(2, description);
20         statement.setString(3,DocumentStatus.DRAFT.toString());
21         statement.setString(4,documentPeriod);
22         statement.setString(5,documentFormat.toString());
23         statement.setString(6,filePath);
24         statement.setInt(7,author.getId());
25         statement.setDate(8, java.sql.Date.valueOf(java.time.
26         LocalDate.now()));
27         statement.setString(9, title);
28         statement.executeUpdate();
29         statement.close();
30         return true;
31     }catch(SQLException e){
32         LOGGER.log(Level.SEVERE, "Errore durante addDocument(
33         authorId=" + (author!=null?author.getId():null) + ")", e);
34         return false;
35     }
36 }
```

Listing 2: Codice del metodo `DocumentDAO.addDocument(...)`

3.3.4 UserDAO

Il `UserDAO` gestisce la persistenza della classe `User`. Fornisce operazioni per la creazione di nuovi utenti, la ricerca di utenti esistenti e l'aggiornamento delle informazioni associate a un profilo.

Questo DAO è utilizzato sia durante le operazioni di autenticazione e registrazione, sia nelle funzionalità amministrative che richiedono la gestione degli utenti, come la promozione a moderatore o l'eventuale rimozione di un account. Quando l'utente manipola le proprie collezioni o i propri preferiti il flusso di esecuzione accede a questa classe, che richiama altre classi DAO quando necessario.

3.3.5 PublishRequestDAO

La classe `PublishRequestDAO` si occupa della gestione delle richieste di pubblicazione rappresentate dalla classe `PublishRequest`. Esso fornisce metodi per la creazione di nuove richieste di pubblicazione di un documento, il recupero delle richieste in attesa e l'aggiornamento dello stato a seguito della valutazione da parte di un moderatore.

Grazie a questo DAO, il processo di pubblicazione dei documenti risulta separato dalla gestione dei documenti stessi, permettendo di mantenere traccia delle decisioni di moderazione e dello storico delle richieste.

3.3.6 CollectionDAO

Il `CollectionDAO` è responsabile della gestione delle collezioni di documenti. Gestisce la creazione e la modifica delle collezioni, nonché l'associazione e la disassociazione dei documenti alle collezioni tramite le opportune tabelle di collegamento.

3.3.7 CommentDAO

`CommentDAO` controlla le funzionalità dei commenti associati ai documenti. Espone metodi per l'aggiunta di nuovi commenti, il recupero dei commenti relativi a un determinato documento o a un determinato utente e la loro eventuale rimozione.

3.3.8 DocumentRelationDAO

Il `DocumentRelationDAO` si occupa della gestione delle relazioni tra documenti rappresentate dalla classe `DocumentRelation`. Fornisce operazioni per la creazione, la ricerca e la rimozione delle relazioni, nonché per l'aggiornamento del loro stato di conferma.

Questo DAO garantisce che, per ogni coppia di documenti, possa esistere al massimo una relazione, in accordo con i vincoli definiti nel modello relazionale. La separazione di questa logica in un DAO dedicato consente di mantenere chiara e coerente la gestione dei collegamenti semantici tra i documenti.

3.3.9 TagDAO

La classe `TagDAO` realizza il livello di accesso ai dati per l'entità di dominio `Tag`, incapsulando tutte le operazioni di persistenza e recupero dei tag dal database.

In particolare, la classe fornisce:

- un metodo di inserimento (`addTag`) per la creazione di nuovi tag persistenti;
- un metodo di ricerca (`findByLabelNormalized`) che consente di recuperare un tag a partire dalla sua etichetta, applicando una normalizzazione case-insensitive e whitespace-insensitive.

3.3.10 TagChangeRequestDAO

La classe `TagChangeRequestDAO` implementa l'accesso ai dati per l'entità di dominio `TagChangeRequest`, incapsulando le query SQL necessarie a creare, aggiornare e consultare le richieste di modifica dei tag.

La creazione di nuove richieste è gestita tramite due metodi distinti, coerenti con le due varianti di `TagChangeRequest`: `addRequestForExistingTag` (operazioni `ADD` o `REMOVE` su tag già esistenti) e `addRequestForNewTag` (proposta di nuovo tag tramite `proposedLabel`).

Il metodo `updateStatus` consente al moderatore di aggiornare lo stato della richiesta, impostando anche `date_result` e salvando l'identificativo del moderatore che ha preso la decisione.

La conversione tra record SQL e oggetto di dominio è centralizzata nel metodo `map`, che ricostruisce le associazioni con `Document` e, se presente, con `User` (moderatore). Infine, `existsPendingDuplicate` permette di verificare l'esistenza di una richiesta `PENDING` equivalente, prevenendo duplicati logici sullo stesso documento e operazione.

4 Testing

Per verificare la correttezza del codice e del flusso di esecuzione delle funzionalità, è ragionevole definire delle classi adibite al testing. Tramite la libreria **JUnit**, sono stati definiti tre livelli di test:

- Testing per la **Business Logic**, dove si verificano i metodi di ogni controller considerando esiti positivi e negativi di ogni funzionalità.
- Testing del **Domain Model**, ristretto soltanto alla classe `DocumentSearchCriteriaBuilder`, poiché unica ad implementare delle funzionalità a livello di dominio differenti da getter e setter.
- Testing per l'**ORM**, dove si interagisce con il database seguendo i metodi delle classi DAO e se ne verifica successivamente lo stato e le tuple aggiunte, rimosse o modificate.

Come riportato nella sezione precedente, il database di produzione non subisce nessuna interazione con queste classi. E' il database creato appositamente per il testing a ricevere le query dei metodi.

Per ogni livello di testing verranno riportati degli esempi, per poi mostrare l'esito di tutti tramite immagini.

4.1 Test Business Logic

Nel Listing 3 troviamo il codice che testa la creazione di un nuovo documento:

```
1  @Test
2  void createDocument() {
3      // crea documento con titolo
4      controller.createDocument("MyDoc", "desc", "1900-1950",
DocumentFormat.PDF, List.of("a", "b"));
5      List<Document> docs = documentDAO.getDocumentsByAuthor(
currentUser.getId());
6      assertFalse(docs.isEmpty());
7      assertEquals("MyDoc", docs.get(0).getTitle());
8
9      // incrementa nextFileName
10     int prev = currentUser.getNextFileName();
11     controller.createDocument("T", "d", "1900", DocumentFormat.PDF,
List.of("t"));
12     assertEquals(prev + 1, currentUser.getNextFileName());
13
14     // input non validi non devono lanciare eccezioni
15     assertDoesNotThrow(() -> controller.createDocument(null, "d", "
1900", DocumentFormat.PDF, List.of("t")));
16     assertDoesNotThrow(() -> controller.createDocument("Doc", null,
"1900", DocumentFormat.PDF, List.of("t")));
17     assertDoesNotThrow(() -> controller.createDocument("Doc", "d",
null, DocumentFormat.PDF, List.of("t")));
18     assertDoesNotThrow(() -> controller.createDocument("Doc", "d", "
1900", null, List.of("t")));
19 }
```

Listing 3: Codice del testing di addDocument()

Mentre nel Listing 4 è presente il test per l'aggiornamento dello status di una PublishRequest da parte di un moderatore:

```
1  @Test
2  void updateDocumentStatus_positive_and_negative_and_auth() {
3      // POSITIVO: crea doc + richiesta pendente, approva tramite
moderatorController
4      documentDAO.addDocument(normalUser, "P", "d", "2000",
DocumentFormat.PDF, "fp", "fn", List.of("t"));
5      int docId = documentDAO.getDocumentsByAuthor(normalUser.getId())
.get(0).getId();
6      publishRequestDAO.addRequest(documentDAO.getDocumentById(docId))
;
7
8      moderatorController.updateDocumentStatus(docId, RequestStatus.
APPROVED);
9
10     // il documento deve essere pubblicato
11     Document published = documentDAO.getDocumentById(docId);
12     assertNotNull(published);
13     assertEquals(DocumentStatus.PUBLISHED, published.getStatus());
14
15     // la richiesta deve risultare APPROVED
16     List<PublishRequest> approvedReqs = publishRequestDAO.
getRequestsByStatus(RequestStatus.APPROVED);
17     assertTrue(approvedReqs.stream().anyMatch(r -> r.getDocument()
!= null && r.getDocument().getId() == docId));
```

```

18
19     // NEGATIVO: documento senza richiesta pendente -> non cambia lo
    status
20     documentDAO.addDocument(normalUser, "NoPending", "d", "2001",
DocumentFormat.TXT, "fp", "fn2", List.of("t"));
21     int docNoPending = documentDAO.getDocumentsByAuthor(normalUser.
getId()).get(0).getId();
22     // non aggiungiamo publishRequest
23     moderatorController.updateDocumentStatus(docNoPending,
RequestStatus.APPROVED);
24     Document notChanged = documentDAO.getDocumentById(docNoPending);
25     assertNotNull(notChanged);
26     assertEquals(DocumentStatus.DRAFT, notChanged.getStatus());
27
28     // AUTH: utente non moderatore prova ad approvare -> non deve
    cambiare lo stato
29     ModeratorController nonMod = new ModeratorController(normalUser)
;
30     // crea doc+request per testare l'accesso negato
31     documentDAO.addDocument(normalUser, "AuthTest", "d", "2002",
DocumentFormat.PDF, "fp", "fn3", List.of("t"));
32     int docAuth = documentDAO.getDocumentsByAuthor(normalUser.getId
()).get(0).getId();
33     publishRequestDAO.addRequest(documentDAO.getDocumentById(docAuth
));
34     nonMod.updateDocumentStatus(docAuth, RequestStatus.REJECTED);
35
36     // essendo non moderatore lo status della richiesta rimane
    PENDING e il documento resta DRAFT
37     List<PublishRequest> pending = publishRequestDAO.
getRequestsByStatus(RequestStatus.PENDING);
38     assertTrue(pending.stream().anyMatch(r -> r.getDocument() !=
null && r.getDocument().getId() == docAuth));
39     assertEquals(DocumentStatus.PENDING, documentDAO.getDocumentById
(docAuth).getStatus());
40 }

```

Listing 4: Codice che testa l'aggiornamento dello status del documento

Il seguente invece si riferisce alla richiesta di aggiunta di un tag in un documento, in questo caso i test sono due per coprire i casi in cui il tag è esistente o nuovo:

```

1 @Test
2 void requestAddExistingTag() {
3     // caso: crea richiesta valida per tag ESISTENTE
4     controller.createDocument("DocTagExist", "d", "1900",
DocumentFormat.PDF, List.of("t"));
5     List<Document> docs = documentDAO.getDocumentsByAuthor(
currentUser.getId());
6     assertFalse(docs.isEmpty());
7     int docId = docs.get(0).getId();
8
9     // creo un tag gi presente in tabella tag
10    String label = "ESISTENTE-" + System.currentTimeMillis();
11    tagDAO.addTag(new Tag(label, "desc"));
12
13    // prima della chiamata nessuna richiesta
14    List<TagChangeRequest> before = tagChangeRequestDAO.getByAuthor(
currentUser.getId());

```

```

15         assertTrue(before.isEmpty());
16
17         // chiamata valida
18         controller.requestAddExistingTag(docId, label);
19
20         // deve esistere UNA richiesta PENDING ADD per quel documento/
tag
21         List<TagChangeRequest> after = tagChangeRequestDAO.getByAuthor(
currentUser.getId());
22         assertEquals(1, after.size());
23         TagChangeRequest r = after.get(0);
24         assertEquals(docId, r.getDocument().getId());
25         assertEquals(TagChangeOperation.ADD, r.getOperation());
26         assertEquals(RequestStatus.PENDING, r.getStatus());
27         assertEquals(label, r.getExistingTagLabel());
28         assertNull(r.getProposedLabel());
29
30         // caso: richiesta DUPLICATA sullo stesso doc/tag -> non crea
nuove righe
31         controller.requestAddExistingTag(docId, label);
32         List<TagChangeRequest> afterDup = tagChangeRequestDAO.
getByAuthor(currentUser.getId());
33         long countSame = afterDup.stream()
34             .filter(x -> x.getDocument() != null
35                 && x.getDocument().getId() == docId
36                 && label.equals(x.getExistingTagLabel())
37                 && x.getOperation() == TagChangeOperation.ADD
38                 && x.getStatus() == RequestStatus.PENDING)
39             .count();
40         assertEquals(1, countSame, "La seconda chiamata non deve creare
richieste duplicate");
41
42         // caso: documento inesistente -> nessuna richiesta aggiuntiva
43         controller.requestAddExistingTag(Integer.MAX_VALUE, label);
44         List<TagChangeRequest> afterInvalidDoc = tagChangeRequestDAO.
getByAuthor(currentUser.getId());
45         long countAfterInvalidDoc = afterInvalidDoc.stream()
46             .filter(x -> x.getDocument() != null
47                 && x.getDocument().getId() == docId
48                 && label.equals(x.getExistingTagLabel())
49                 && x.getOperation() == TagChangeOperation.ADD
50                 && x.getStatus() == RequestStatus.PENDING)
51             .count();
52         assertEquals(1, countAfterInvalidDoc, "Chiamate con docId
invalido non devono modificare il DB");
53     }
54
55     @Test
56     void requestAddNewTag() {
57         // Arrange
58         controller.createDocument(
59             "DocNewTag",
60             "desc",
61             "1900",
62             DocumentFormat.PDF,
63             List.of()
64         );
65

```

```

66     List<Document> docs = documentDAO.getDocumentsByAuthor(
currentUser.getId());
67     assertEquals(1, docs.size());
68     Document doc = docs.get(0);
69
70     String proposedLabel = "NuovoTagSpeciale";
71
72     // Il tag NON deve esistere nel catalogo prima
73     assertNull(tagDAO.findByLabelNormalized(proposedLabel));
74
75     // Act
76     controller.requestAddNewTag(doc.getId(), proposedLabel);
77
78     // Assert
79     List<TagChangeRequest> reqs = tagChangeRequestDAO.
getRequestsByDocument(doc.getId());
80     assertEquals(1, reqs.size());
81     TagChangeRequest r = reqs.get(0);
82
83     assertEquals(TagChangeOperation.ADD, r.getOperation());
84     assertEquals(RequestStatus.PENDING, r.getStatus());
85     assertEquals(doc.getId(), r.getDocument().getId());
86
87     assertNull(r.getExistingTagLabel());
88     assertEquals(proposedLabel, r.getProposedLabel());
89
90     // Il Tag vero non deve ancora esistere
91     assertNull(tagDAO.findByLabelNormalized(proposedLabel));
92 }

```

Listing 5: Codice del testing di requestAddNewTag() e requestAddExistingTag()

✓ BusinessLogic	3 sec 612 ms
✓ AdminControllerTest	1 sec 437 ms
✓ documentsByStatus()	463 ms
✓ getCommentByAuthor()	94 ms
✓ searchUserByEmail()	78 ms
✓ getCommentsByDocument()	79 ms
✓ setModerator()	62 ms
✓ documentsApprovedByModerator()	65 ms
✓ deleteCollection()	69 ms
✓ removeUser()	63 ms
✓ allCollections()	62 ms
✓ documentsByAuthor()	59 ms
✓ allUsers()	50 ms
✓ searchUserById()	53 ms
✓ allDocuments()	169 ms
✓ removeComment()	71 ms
✓ RelationControllerTest	514 ms
✓ addRelation()	99 ms
✓ updateType()	68 ms
✓ searchSource()	62 ms
✓ setConfirmed()	56 ms
✓ searchDestination()	60 ms
✓ daoByConfirm()	62 ms
✓ selected()	48 ms
✓ removeRelation()	59 ms

Figura 16: I test di AdminController e RelationController

✓ UserControllerTest	959 ms
✓ favouritesDocuments()	61 ms
✓ requestAddNewTag()	58 ms
✓ favouritesCollections()	59 ms
✓ removeDocumentToCollection()	68 ms
✓ searchDocuments()	64 ms
✓ requestAddExistingTag()	86 ms
✓ writeComment()	65 ms
✓ addDocumentToCollection()	178 ms
✓ requestRemoveTag()	67 ms
✓ viewOwnDocuments()	58 ms
✓ askForPublication()	104 ms
✓ createDocument()	91 ms
✓ ModeratorControllerTest	437 ms
✓ updateDocumentStatus_positive_and_negat	76 ms
✓ decideTagRequest_positive_negative_and_	95 ms
✓ viewTagRequestHistory()	56 ms
✓ viewRequestHistory()	67 ms
✓ viewPendingDocumentRequests_positive_ar	71 ms
✓ viewPendingTagRequests_positive_and_aut	72 ms
✓ LoginControllerTest	265 ms
✓ register()	164 ms
✓ emailAvailable()	48 ms
✓ login()	53 ms

Figura 17: I test di UserController, ModeratorController e LoginController

4.2 Test Domain Model

Come già detto, nel package Domain Model la maggior parte delle classi contiene solo metodi getter e setter. Questo perché in questo layer si depositano solo le funzionalità di dominio, che dipende dalla natura del software e del contesto rappresentato da esso. In questo caso quindi solo `DocumentSearchCriteriaBuilder` incorpora funzionalità di dominio da dover testare:

```

1 @Test
2     // Verifica che, quando si impostano tutti i campi, il criterio
   risultante contenga
3     // i valori corrispondenti e che venga fatta una copia difensiva
   della lista dei tag.
4     void buildAllFields() {
5         Date after = new Date(1000L);
6         Date before = new Date(2000L);
7         List<String> tags = new ArrayList<>(Arrays.asList("tag1", "tag2"
   ));
8
9         DocumentSearchCriteria criteria = DocumentSearchCriteriaBuilder.
   getInstance()
10             .setDocumentTitle("My doc")
11             .setAuthorId(5)
12             .setFormat(DocumentFormat.PDF)
13             .setCreatedAfter(after)
14             .setCreatedBefore(before)
15             .setTags(tags)
16             .build();
17
18         assertTrue(criteria.getDocumentTitle().isPresent());
19         assertEquals("My doc", criteria.getDocumentTitle().get());
20

```

```

21     assertTrue(criteria.getAuthorId().isPresent());
22     assertEquals(5, criteria.getAuthorId().get());
23
24     assertTrue(criteria.getFormat().isPresent());
25     assertEquals(DocumentFormat.PDF, criteria.getFormat().get());
26
27     assertTrue(criteria.getCreatedAfter().isPresent());
28     assertEquals(after, criteria.getCreatedAfter().get());
29
30     assertTrue(criteria.getCreatedBefore().isPresent());
31     assertEquals(before, criteria.getCreatedBefore().get());
32
33     assertTrue(criteria.getTags().isPresent());
34     assertEquals(2, criteria.getTags().get().size());
35     assertEquals(Arrays.asList("tag1", "tag2"), criteria.getTags().
get());
36
37     // ensure defensive copy: modifying original list does not
affect criteria
38     tags.add("newtag");
39     assertEquals(2, criteria.getTags().get().size());
40 }
41
42 @Test
43 // Verifica che, se non si impostano campi, il criterio restituisca
Optionals vuoti
44 void buildNoFields() {
45     DocumentSearchCriteria criteria = DocumentSearchCriteriaBuilder.
getInstance()
46         .build();
47
48     assertFalse(criteria.getDocumentTitle().isPresent());
49     assertFalse(criteria.getAuthorId().isPresent());
50     assertFalse(criteria.getFormat().isPresent());
51     assertFalse(criteria.getCreatedAfter().isPresent());
52     assertFalse(criteria.getCreatedBefore().isPresent());
53     assertFalse(criteria.getTags().isPresent());
54 }

```

Listing 6: Codice del testing DocumentSearchCriteriaBuilder.build())

E vengono riportati gli esiti:

✓ DomainModel	26 ms
✓ DocumentSearchCriteriaBuilderTest	26 ms
✓ buildNoFields()	23 ms
✓ buildAllFields()	3 ms
✓ setTagsNull()	

Figura 18: I test di DocumentSearchCriteriaBuilder

4.3 Test ORM

Nel Listing 7 è presente il codice di testing per la funzionalità di aggiunta e rimozione di una relazione tra due documenti. Il testing viene fatto a livello di operazione CRUD a

database

```
1 @Test
2     void addAndRemoveDocumentRelation() {
3         // ensure no relation
4         assertEquals(0, countRelationRows(srcDocId, dstDocId));
5
6         relationDAO.addDocumentRelation(srcDocId, dstDocId,
7         DocumentRelationType.QUOTE, false);
8         assertEquals(1, countRelationRows(srcDocId, dstDocId));
9
10        relationDAO.removeDocumentRelation(srcDocId, dstDocId);
11        assertEquals(0, countRelationRows(srcDocId, dstDocId));
12    }
```

Listing 7: Codice del testing

Nel seguente Listing invece viene mostrato a livello di operazione CRUD il test della rimozione di un commento:

```
1 @Test
2     void removeComment() {
3         String uniqueText = "cmt-rem-" + System.currentTimeMillis();
4         commentDAO.addComment(uniqueText, testUser.getId(), testDocId);
5
6         List<Comment> comments = commentDAO.getCommentsByDocument(
7         testDocId);
8         Comment added = comments.stream().filter(c -> uniqueText.equals(
9         c.getText())).findFirst().orElse(null);
10        assertNotNull(added, "Commento aggiunto non trovato prima della
11        rimozione");
12
13        commentDAO.removeComment(added.getId());
14
15        List<Comment> after = commentDAO.getCommentsByDocument(testDocId
16        );
17        assertFalse(after.stream().anyMatch(c -> uniqueText.equals(c.
18        getText())));
19    }
```

Listing 8: Test Remove comment

Nelle immagini successive vengono mostrati tutti i test ed esiti delle classi del package ORM:

✓ ORM	3 sec 949 ms
✓ PublishRequestDAOTest	704 ms
✓ removeRequest()	533 ms
✓ updateRequestStatusAndQueries()	79 ms
✓ addRequestAndGetPending()	92 ms
✓ TagChangeRequestDAOTest	1 sec 33 ms
✓ addRequestForExistingTag_withNewLabelRequest_doesNotInsert()	70 ms
✓ existsPendingDuplicate_returnsTrueForPendingExistingTagRequest()	74 ms
✓ updateStatus_withNonExistingId_doesNotChangeAnything()	189 ms
✓ getRequestByModerator_returnsOnlyRequestsOfGivenModerator()	88 ms
✓ getById_existingId_returnsMappedRequest()	76 ms
✓ getRequestsByDocument_returnsOnlyRequestsForThatDocument()	78 ms
✓ addRequestForNewTag_persistsRequestAndSetsId()	73 ms
✓ addRequestForExistingTag_persistsRequestAndSetsId()	73 ms
✓ getPending_returnsOnlyPendingRequests()	79 ms
✓ updateStatus_updatesStatusDateAndModerator()	80 ms
✓ getByAuthor_returnsOnlyRequestsOfDocumentsByThatAuthor()	85 ms
✓ existsPendingDuplicate_returnsFalseWhenStatusIsNotPending()	68 ms
✓ TagDAOTest	207 ms
✓ addTag_thenFindBySameLabel_returnsTag()	47 ms
✓ findByLabelNormalized_onEmptyTable_returnsNull()	49 ms
✓ findByLabelNormalized_ignoresCaseAndWhitespace()	56 ms
✓ findByLabelNormalized_returnsNullForDifferentLabel()	55 ms

Figura 19: I test PublishRequestDAO, TagChangeRequestDAO, TagDAO

✓ DocumentDAOTest	497 ms
✓ deleteDocument()	72 ms
✓ getDocumentsByAuthor()	69 ms
✓ searchDocuments()	78 ms
✓ getDocumentsByStatus()	63 ms
✓ getAllDocuments()	51 ms
✓ addDocument()	53 ms
✓ updateDocumentStatus()	54 ms
✓ getDocumentById()	57 ms
✓ CollectionDAOTest	408 ms
✓ addCollection()	59 ms
✓ getAllCollectionsAndGetById()	89 ms
✓ addAndRemoveDocumentToCollection()	86 ms
✓ deleteCollection()	174 ms
✓ CommentDAOTest	125 ms
✓ addAndGetCommentsByDocumentAndAuthor()	58 ms
✓ removeComment()	67 ms

Figura 20: I test di DocumentDAO, CollectionDAO, CommentDAO

✓	✓	UserDAOTest	506 ms
	✓	addAndRemoveFavouriteDocument()	75 ms
	✓	getUserByEmail()	61 ms
	✓	getAllUsers()	54 ms
	✓	addAndRemoveFavouriteCollection()	60 ms
	✓	setModeratorAndGetModerators()	59 ms
	✓	addAndGetUser()	56 ms
	✓	removeUser()	83 ms
	✓	updateNextFileName()	58 ms
✓	✓	DocumentRelationDAOTest	469 ms
	✓	updateDocumentRelation()	64 ms
	✓	getRelationsByConfirmAndSetConfirmed()	85 ms
	✓	addAndRemoveDocumentRelation()	58 ms
	✓	getSourceAndDestinationRelationDocument()	64 ms
	✓	duplicateInsertShouldNotCreateMultipleRelations()	59 ms
	✓	daoDirectConfirmBehavior()	67 ms
	✓	getAllSourceAndAllDestinationRelationDocument()	72 ms

Figura 21: I test di UserDAO e DocumentRelationDAO