Tivity

Home for creator's next generation

Niccolò Zanieri

Febbraio 2022

Indice

1 Introduzione

Tivity è il nome che ho deciso di dare al progetto da me realizzato in quanto questo pone le basi per lo sviluppo di un social network dedicato ai creatori di contenuti digitali, per i quali la parola *Creativity* è di fondamentale importanza.

Lo scopo dell'applicazione è quello di fornire ai creatori di contenuti digitali un posto nel quale mostrare i propri lavori, trovare ispirazione dal lavoro altrui ed avere a disposizione una vetrina che fornisce maggior visibilità rispetto ad altri tipi di piattaforma.
Inoltre, l' applicazione vorrebbe permettere a clienti alla ricerca di professionisti del settore di accedere ad un catalogo di figure professionali di diverso tipo sulla base di interessi e necessità.
È possibile consultare lo statement del progetto nell'Appendice A.

Allo stato attuale del progetto, questo è composto di una interfaccia grafica tramite la quale è possibile effettuare l'iscrizione alla piattaforma, nel caso l'utente non disponga di un profilo esistente, oppure accedere direttamente se l'utente è già registrato.
Tramite l'interazione con un database locale contenente i dati degli utenti registrati, l'applicazione è in grado di distinguere se si stanno utilizzando delle credenziali valide in caso di accesso e se, in fase di iscrizione, si sta tentando di creare un profilo con un nome utente già esistente.
Eventuali errori nelle procedure sopracitate sono gestiti andando a mostrare opportuni segnali di errore e fornendo la possibilità di ripetere la procedura non andata a buon fine.

Infine, nel caso l'utente riesca ad accedere senza problemi viene presentata una pagina di benvenuto personalizzata con lo username del profilo con il quale si è effettuato l'accesso.

2 Analisi

2.1 Casi d'uso

In fase di analisi delle responsabilità dell'applicazione ho individuato due principali aree di interesse per i casi d'uso: la prima legata all'interazione diretta di un utente con il sistema e la seconda associata all'interazione tra cliente e creator. La seconda area risulta essere meno sviluppata in quanto attualmente ho deciso di concetrarmi maggiormente sulla prima.
In questa sezione ho inserito soltanto gli use-case template relativi ai casi d'uso implementati, i template rimanenti si possono trovare all'interno dell'Appendice B.
Di seguito sono riportati gli use-case diagram associati a queste aree.

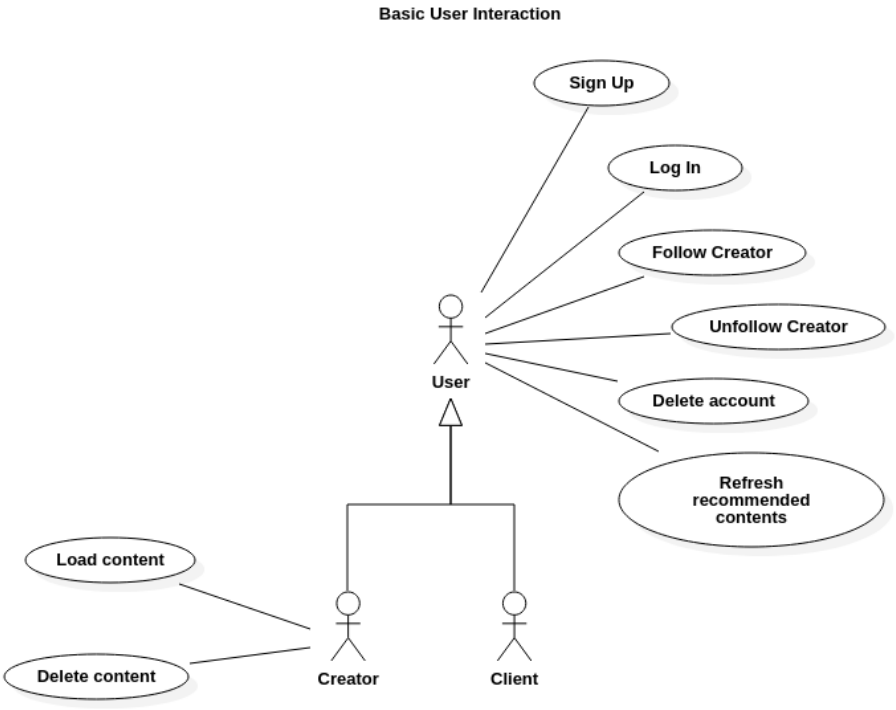


Figura 2.1: Casi d'uso relativi all'interazione dell'utente con l'applicazione.

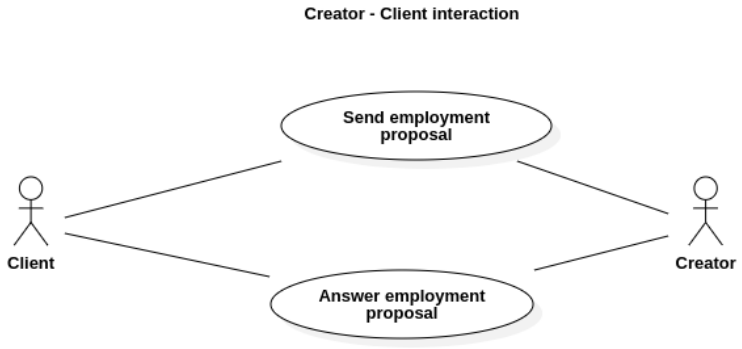


Figura 2.2: Casi d'uso relativi all'interazione tra cliente e creator.

2.1.1 Log In

Come già descritto in precedenza, allo stato attuale l'applicazione realizza solo alcuni dei casi d'uso rappresentati nei diagrammi e questo è senz'altro uno dei più rilevanti per il progetto.
Si può facilmente intuire dal nome che lo scopo del caso d'uso sia quello di permettere l'accesso solo ad utenti registrati: una descrizione più dettagliata si trova in Figura 2.3.

Nota: ho deciso di realizzare i casi d'uso in inglese nonostante questa relazione sia in italiano perché mi risulta più semplice progettare e scrivere codice in inglese che ritengo una lingua più pragmatica da questo punto di vista.

Use-case field	Description
Use-case name	Log In
Subject area	Basic User Interaction
Actors	User
Preconditions	The User is in the log in page
Main success scenario	1. User enters username and password 2. User submits the log in request 3. System recognizes credentials as valid 4. System loads the app's main page
Alternative scenarios	2.a System detects invalid credentials 3.a System shows error message and allows User to try log in again, returns to MSS at step 1

Figura 2.3: Use-case template del caso d'uso Log In

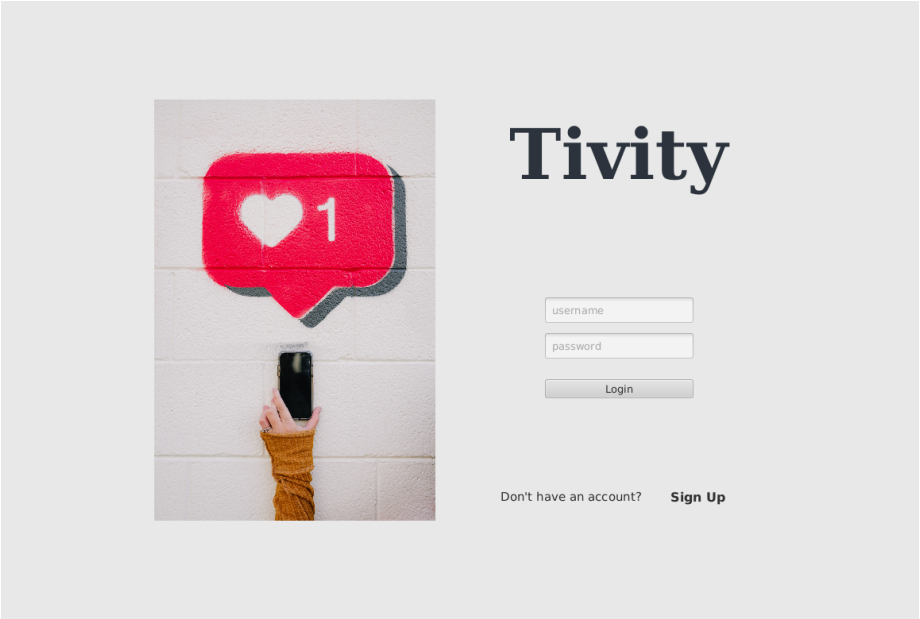
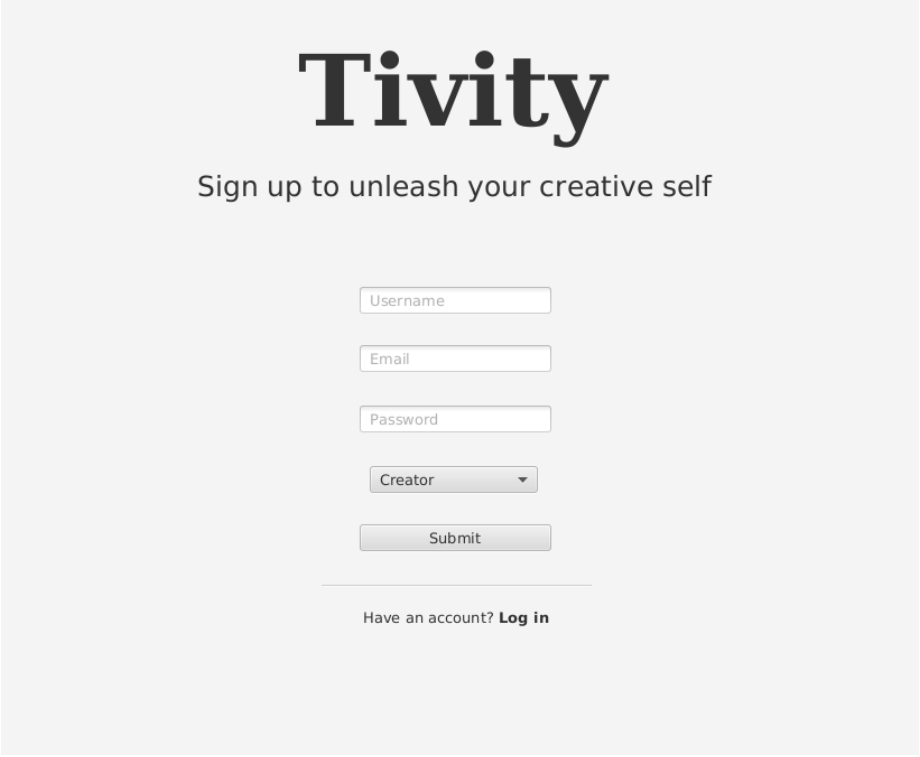


Figura 2.4: Pagina di log in di Tivity

2.1.2 Sign Up

Altro caso d'uso fondamentale è ovviamente quello del sign up che ci permette di aggiungere utenti all'applicazione.
Alla pagina successiva si trovano il template del caso e la visualizzazione della pagina di sign up dell'applicazione.

Use-case field	Description
Use-case name	Sign Up
Subject area	Basic User Interaction
Actors	User
Preconditions	The User is in the sign up page
Main success scenario	1. User enters username, email, password he wants for his profile 2. User selects the kind of profile he wants to create, either Creator or Client 3. User submits the compiled form 4. System adds User's profile to the database 5. System shows a message to confirm the successful sign up
Alternative scenarios	4.a System detects duplicated username 5.a System shows error message to User
Garantee	The database is kept in a consistent state. In case of success a profile with the desired credentials is added and can be used to access the application.



2.2 Modello di dominio

Lo svolgimento del progetto mi ha portato alla produzione di due class diagram i cui scopi e motivi di esistere sono ben diversi. Prima di procedere alla scrittura del codice ho realizzato un diagramma a partire dallo statement iniziale per andare a dare una descrizione del dominio di interesse dell'applicazione. Mentre, una volta terminata la stesura del codice mi è apparso naturale dover includere un ulteriore class diagram che rappresentasse ciò che ho effettivamente realizzato: in quanto questo risulta essere diverso dalla descrizione iniziale che, seppur orientata all'implementazione, non teneva di conto del mantenimento della persistenza dei dati e dell'interazione con l'utente.

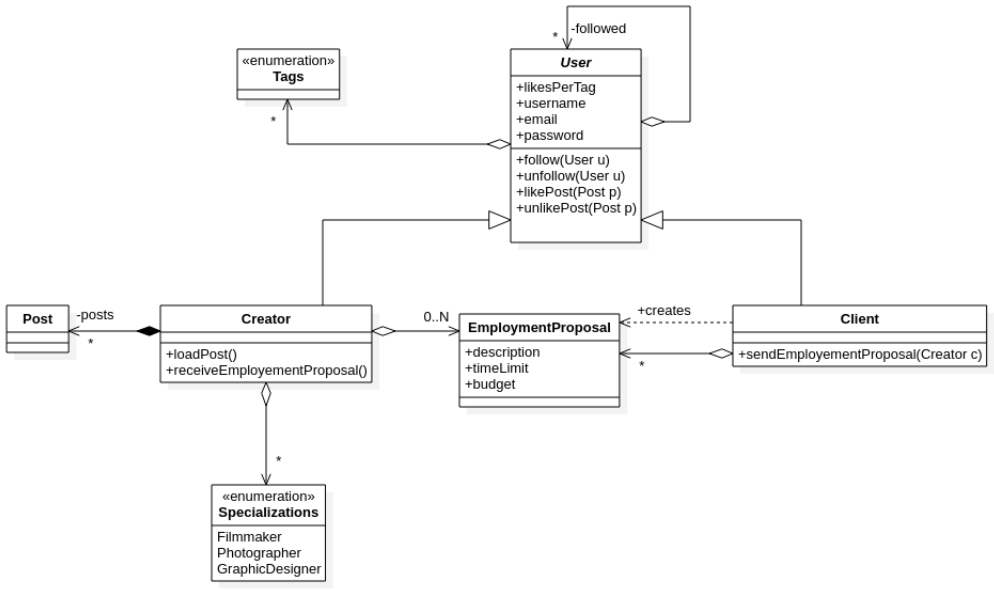


Figura 2.5: Class diagram iniziale orientato all'implementazione

A causa delle elevate dimensioni del diagramma realizzato in seguito all'implementazione ho deciso di inserirlo all'interno dell'Appendice C per poi andare a descrivere qui di seguito alcune sue sottoparti.

2.3 Design Pattern

2.3.1 MVC

Uno degli scopi fondamentali per il quale ho deciso di realizzare questo progetto era quello di produrre una applicazione che permettesse all'utente di interagire con il sistema tramite un interfaccia grafica e di avere un livello di persistenza dei dati.

A tale scopo ho deciso di utilizzare **JavaFx** per la realizzazione dell'interfaccia grafica e questo ha portato alla scelta dell'utilizzo del patter MVC. Il Model View Controller è un pattern che permette di separare i livelli di presentazione dei dati all'utente, descrizione dei dati presentati e mediazione tra le azioni dell'utente ed i dati dell'applicazione.
In particolare, JavaFx permette in modo molto facile di realizzare il pattern andando a fornire la possibilità di creare viste con file di tipo *.fxml* che sono facilmente personalizzabili tramite l'uso dell'applicazione SceneBuilder e di connettere queste con dei controller tramite lo stesso programma. Infine, quest'ultimi hanno facilmente accesso al modello se questo si trova nello stesso package.

Purtroppo, JavaFx richiede che i controller e le viste si trovino in particolari posizioni all'interno del progetto per permettere una semplice integrazione con SceneBuilder e per questo non compare nessun package che racchiuda tutti i controller. Per lo stesso motivo non sono riuscito a separare i file *.fxml* dai fogli di stile usati per parte della loro personalizzazione.

2.3.2 Data Access Object

Come evidenziato di sopra, uno degli aspetti che più mi interessava implementare è quello del mantenimento della persistenza dei dati. Dopo aver valuto alcune possibili opzioni, ho optato per utilizzare il pattern Data Access Object: un pattern che permette di isolare l'accesso ad un database in questo caso relazionale dalle responsabilità assolte dalle classi del modello.
Attualmente, quello che ho fatto è stato creare due classi il cui scopo è quello di fornire metodi per andare ad interagire con un database Postgresql installato sul mio calcolatore.
Come si può osservare in Figura 2.6, le classi CreatorDAO e ClientDAO si occupano dell'interazione con il database esponendo dei metodi per inserire o rimuovere dati relativi a Creator o Client. Oltre a questo, le due classi permettono di ottenere oggetti dei rispettivi tipi utilizzando uno username valido (vedere il codice per la signature completa) e di aggiungere o rimuovere follower per un certo Creator.

Queste due classi sono di fondamentale importanza in quanto è con esse che i controller del MVC interagiscono per avere accesso ai dati dell'applicazione. È possibile riscontrare la presenza di questa interazione nel class diagram dell'Appendice C.

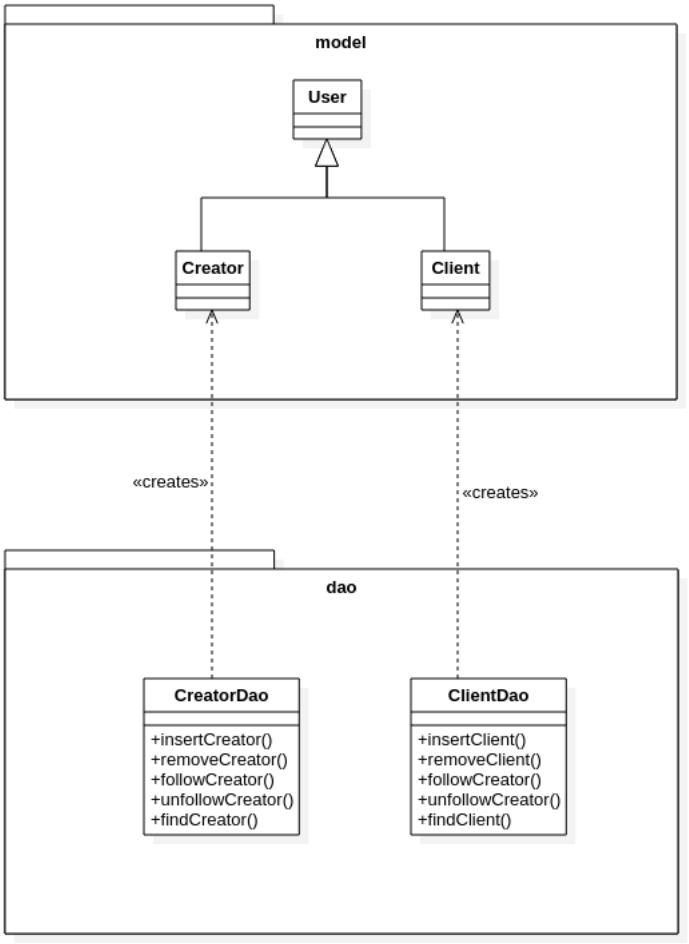


Figura 2.6: Class diagram che descrive le classi che realizzano il pattern DAO.

2.3.3 ConnectionPool e Singleton

Un aspetto fondamentale dell'interazione del codice con un database è quello della gestione delle connessioni con quest'ultimo. A seguito delle risorse consigliatemi dal professor Vicario ho deciso di implementare la pratica del *Connection Pooling*: lo scopo principale della pratica è quello di ridurre il più possibile il numero di richieste di connessione fatte ad un dbms favorendo il riutilizzo di connessioni già aperte.

L'idea di base è quella di realizzare tutte le connessioni al momento della creazione dell'oggetto di tipo ConnectionPool che poi permette ad altre classi di riutilizzare queste connessioni già esistenti e di rilasciarle una volta terminato l'uso del dbms.
Per maggiori dettagli rimando alla lettura dell'articolo presente al link "https://www.baeldung.com/java-connection-pooling" che ho usato per la mia implementazione.
Oltre a questo, ho deciso di implementare la classe ConnectionPool come un Singleton per far sì che tutti i controller di una stessa sessione dell'applicazione facciano accesso ad una stessa istanza della classe ConnectionPool.
Per come ho realizzato la *pool*, questa inizializza dieci connessioni al momento dell'istanziazione e andare a creare un oggetto di tipo ConnectionPool ogniqualvolta un controller avesse avuto bisogno di accedere al dbms avrebbe eliminato del tutto il vantaggio dato dall'uso del pattern.
Inoltre, l'uso del Singleton mi ha permesso di evitare di dover creare un oggetto di tipo ConnectionPool nell'applicazione principale e passarlo ai vari metodi ogni volta che fosse necessario rendendo il codice più pulito e riducendo la possibilità di errori.

2.4 DBMS

Per gestire i dati in modo persistente ho optato, come già introdotto in precedenza, per l'utilizzo di un database gestito con l'RDBMS Posgresql. Di seguito in Figura 2.7 il diagramma ER che ho realizzato prima di definire uno schema del database nel modello relazionale. Lo schema del modello relazionale ottenuto si trova all'Appendice D.

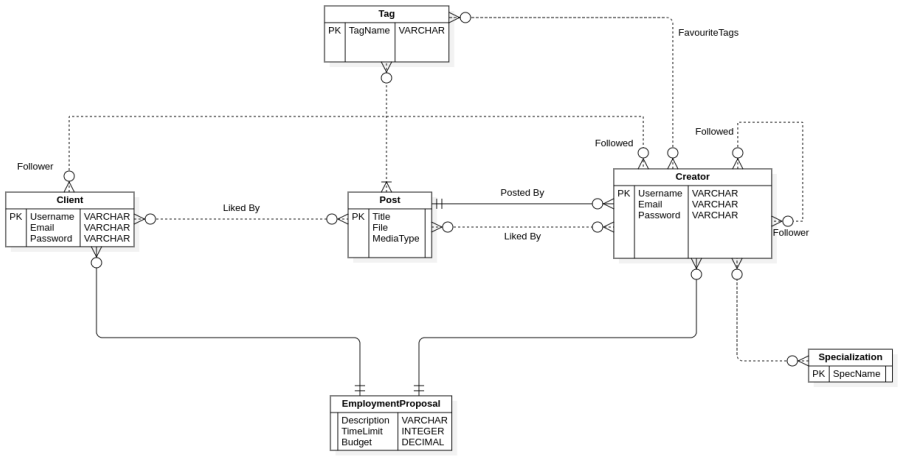


Figura 2.7: Diagramma Entity Relationship del dominio di interesse

3 Implementazione delle classi

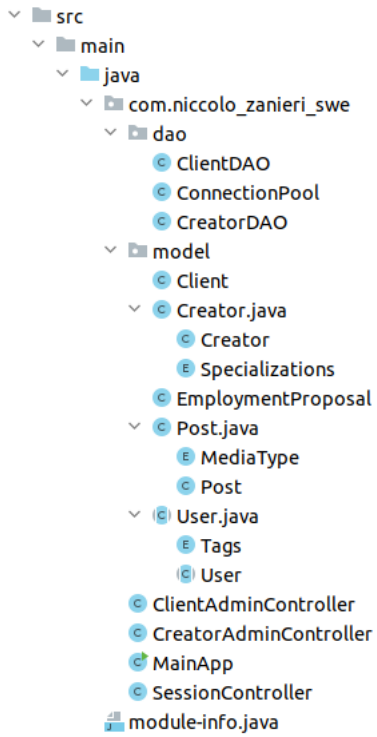


Figura 3.1: Struttura delle classi e dei package del progetto descritto

3.1 Model

Il package `model` contiene tutte quelle classi che vanno a descrivere le entità presenti nel dominio di interesse e sono state implementate a partire dal diagramma delle classi presente in Figura 2.5.

3.1.1 User

Classe astratta che rappresenta l'utente dell'applicazione e che si occupa di andare a rappresentare e gestire tutte quegli aspetti comuni ai vari tipi di user.
La classe definisce che un qualsiasi utente deve essere descritto tramite uno username, una mail ed una password oltre a mantenere informazioni riguardo i gusti personali e i Creator seguiti.

3.1.2 Client

La classe va a descrivere tutte le informazioni di interesse relative ad un profilo di tipo Cliente.
Buona parte delle informazioni sono gestite a livello della superclasse User mentre, in aggiunta, questa classe permette l'invio di proposte di lavoro ai Creator.
L'insieme delle proposte di lavoro inviate è memorizzato all'interno dell'attributo `proposals`.

3.1.3 Creator

Anche la classe Creator estende la classe astratta User andando ad aggiungere informazioni relative alla specializzazione del creator in questione e alla gestione delle proposte di lavoro.

3.1.4 EmploymentProposal

Questa classe racchiude le informazioni minimali per la descrizione di una proposta di lavoro che, come emerso dallo statement all'Appendice A, è caratterizzata da una durata temporale, un bugdet e una descrizione testuale.

3.1.5 Post

Come nel caso delle proposte d'impiego, la classe si limita a riportare le informazioni principali per la descrizione di un post: nome del file caricato, descrizione, tipologia del file multimediale.

3.2 Controller

Nonostante non esista nel progetto un package dedicato per i controller del MVC per motivi già riportati, ho deciso di descriverli in una sezione apposita in quanto logicamente separati dal modello.

3.2.1 ClientAdminController

Questo controller ha lo scopo di gestire quelle operazioni di gestione delle informazioni relative ai Client all'interno della base di dati.
La classe permette di aggiungere le informazioni di un profilo cliente al database e di recuperare le informazioni relative ad un dato nome utente se e solo se viene fornita la password corretta.
Al livello attuale dell'applicazione sono gestite soltanto le informazioni di base degli utenti quali lo username, la mail e la password.

3.2.2 CreatorAdminController

Classe del tutto analoga a quella precedente ma che si occupa della gestione delle informazioni dei Creator.

3.2.3 SessionController

Ad ora questo è il controller più consistente del progetto. Lo scopo di questa classe è quello di gestire le sessioni di accesso degli utenti andandosi ad occupare di eventuali richieste di *log in* e/o *sign up*.
È all'interno di questa classe che viene gestito il passaggio tra le scene e la presentazione di eventuali segnali di errore o di successo delle operazioni.

3.3 Viste

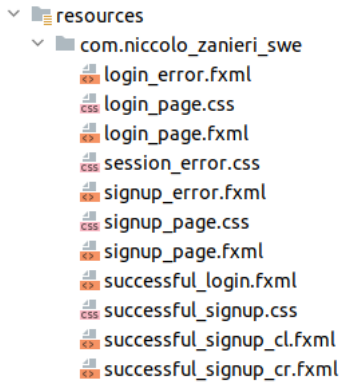


Figura 3.2: Struttura dei file relativi alle viste

Per quanto riguarda le viste, come accennato parlando del pattern MVC, ho deciso di utilizzare **JavaFx** che fornisce la possibilità di realizzare viste in modo molto semplice tramite l'applicazione SceneBuilder.
In breve, SceneBuilder consente di modificare tramite una interfaccia grafica dei file *.fxml* i quali, tramite un linguaggio di mark-up, permettono di creare interfacce grafiche per le proprie applicazioni.
Pertanto, i file strettamente relativi alle viste sono quelli con estensione *.fxml* e credo che i nomi che possiedono siano già abbastanza esplicativi dello scopo dei vari file.

4 Test

4.1 Model

4.1.1 ClientTest

All'interno di questa suite si trovano soltanto un metodo per inizializzare un oggetto di tipo Client ed un metodo che testa se le proposte di impiego sono inviate correttamente ad un Creator. In particolare, si verifica se il metodo `sendEmploymentProposal` restituisce falso nel caso in cui il Creator abbia già raggiunto il massimo numero di richieste.

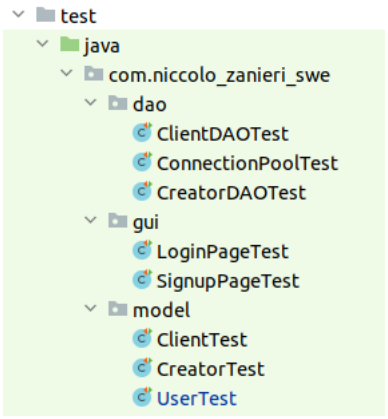


Figura 4.1: Struttura delle classi e dei package relativi al test del progetto

4.1.2 CreatorTest

Questa classe contiene due test: il primo verifica il corretto funzionamento del costruttore ed il secondo si occupa di valutare se il metodo `receiveProposal` aggiunge le proposte quando possibile e restituisce falso quando si supera il limite massimo di proposte.

4.1.3 UserTest

Il metodo `UserCtorTest` contenuto nella classe `UserTest` ha lo scopo di verificare il corretto funzionamento del costruttore della classe User: in particolare, questo si occupa di verificare che il numero di like per tag sia zero per tutti i tag e che i tag preferiti siano inseriti correttamente.
Per quanto riguarda il metodo `UserIsFollowedTest`, questo si asssicura che il metodo usato per valutare se un Creator è già seguito o meno funzioni come atteso.
Infine, `UserFollowTest` va a verificare che quando un utente inizia a seguire un Creator non può fare una nuova richiesta e "seguirlo due volte".

4.2 DAO

4.2.1 ClientDAOTest

Oltre ai metodi per il setup ed il teardown che si occupano di gestire la *pool* per l'accesso a alle connessioni al database, questa suite contiene 5 metodi che verificano la corretta esecuzione di: inserimento di un cliente nel database, rimozione di un cliente dalla base dati usando il suo nome utente, inserimento o rimozione delle informazioni riguardo la relazione "seguace-seguito" e recupero di tutte le informazioni di un cliente tramite il suo username.
Ovviamente i test si occupano di provare i metodi anche in casi di eccezione.

4.2.2 CreatorDAOTest

L'insieme dei test presenti in questa classe è del tutto analogo a quelli descritti alla sottosezione precedente con l'eccezione che riguardo creatori e non clienti.

4.2.3 ConnectionPoolTest

Ai fini di testare il corretto funzionamento del *connection pooling* ho utilizzato una apposita suite di test.
In questa classe viene testata la corretta inizializzazione della *pool*, il metodo usato per ottenere una connessione al database e due metodi usati per il rilascio di connessioni.

4.3 GUI

Come si può intuire dai nomi delle classi presenti in questa cartella, i test qui presenti si occupano di testare il corretto funzionamento dell'interfaccia grafica realizzata in relazione a due casi d'uso: quello relativo al log in e quello relativo al sign up di un utente.
Le due classi presenti vanno a valutare se l'interazione dell'utente con l'applicazione rispetta le regole della logica del modello assicurandosi che vengano mostrati opportunamente segnali di successo o di errore.
Oltre a ciò, le suite verificano che il passaggio tra le varie scene avvenga come ci si aspetta e che in caso di sign up i dati inseriti vengano correttamente memorizzati all'interno del database.

A Statement iniziale

Di seguito sono riportati i punti individuati nell'iniziale formulazione dello statement:

- Si vuole realizzare un semplice social network per creatori di contenuti digitali.
- Nell'applicazione, ogni Creator deve avere la possibilità di caricare contenuti da lui prodotti cosi che questi siano visibili agli altri utenti. Inoltre, il Creator deve avere la possibilità di vedere parte dei contenuti caricati dagli altri utenti per i quali potrà indicare il suo gradimento tramite un 'like'.
- I contenuti visualizzati da ogni Creator sono personalizzati e generati ogni volta che il Creator fa richiesta di caricare la pagina principale del social network. Quest'ultima sarà separata dalla pagina usata per caricare i contenuti.
- Ogni Creator per avere accesso all'applicazione necessita di registrarsi inserendo i dati e il tipo di contenuto che produce: i possibili profili professionali sono quello di Fotografo, Filmmaker, Grafico.
- Al momento dell'iscrizione sarà chiesto al creator di inserire il proprio username, la e-mail e una password che dovrà utilizzare per accedere all'applicazione. Oltre a questo verrà richiesto di scegliere, facoltativamente, tra un certo numero di tag che saranno usati per produrre i contenuti personalizzati.
- Inoltre, si vuole che Creator abbia la possibilità di 'seguire' altri Creator, questa scelta aumenta la probabilità di visualizzare contenuti dei profili seguiti rispetto a quelli non seguiti.
- L'applicazione all'avvio chiederà se si vuole accedere inserendo le proprie credenziali oppure se si vuole creare un nuovo profilo.
- Il profilo creato non è limitato alla figura del Creator, oltre a questo si vuole avere la possibilità di creare un profilo di tipo Cliente. Gli utenti registrati con un profilo cliente hanno la possibilità di visualizzare contenuti personalizzati in modo del tutto analogo ai Creator, ciò che li differenzia tuttavia è la possibilità di inviare proposte di lavoro ai Creator.
- Ogni utente di tipo Creator potrà supportare un numero massimo di clienti da lui stesso stabilito al momento dell'iscrizione mentre gli utenti Cliente possono fare un numero indefinito di proposte di lavoro.
- Le richieste di lavoro devono essere compilate dal Cliente inserendo una descrizione del lavoro richiesto, il tempo massimo che è disposto ad attendere per la consegna finale ed una proposta di budget che sarà messo a disposizione.

B Use case template

B.1 Basic User Interaction

Use-case field	Description
Use-case name	Follow creator
Subject area	Basic User Interaction
Actors	User, Creator
Preconditions	The User logged in successfully and has access to the infos about the Creator he wants to follow
Main success scenario	1. User submits a request to follow Creator 2. System verifies that Creator is not already followed 3. System adds Creator to User's followed accounts and adds the info in the database 4. System add User to Creator followers
Alternative scenarios	2.a System detects that Creator is already followed by User 3.a System shows a message saying that User can't follow an already followed Creator
Garantee	The database is kept in a consistent state. A Creator is not followed two times by the same User

Use-case field	Description
Use-case name	Unfollow creator
Subject area	Basic User Interaction
Actors	User, Creator
Preconditions	The User logged in successufully and has access to the infos about the Creator he wants to unfollow. The User is following the Creator
Main success scenario	5. User submits a request to unfollow Creator 6. System removes Creator from User's followed accounts 7. System remove User from Creator followers
Garantee	The database is kept in a consistent state.

Use-case field	Description
Use-case name	Delete account
Subject area	Basic User Interaction
Actors	User
Preconditions	The User has a valid account and logged in successfully
Main success scenario	1. User submits a request to have his account deleted 2. System asks User to enter his password 3. User enters the correct password 4. System deletes User's account from database
Alternative scenario	3.a User enters incorrect password 4.a System show error message, returns to MSS 2
Garantee	The database is kept in a consistent state. User can't access the app with the deleted profile anymore.

Use-case field	Description
Use-case name	Refresh recommended contents
Subject area	Basic User Interaction
Actors	User
Preconditions	The User has a valid account and logged in successfully
Main success scenario	1. User requests to view new recommended contents 2. System decide what contents to show based on User's followed accounts and preferences 3. System shows User the new selected contents

Use-case field	Description
Use-case name	Load content
Subject area	Basic User Interaction
Actors	Creator
Preconditions	The Creator has a valid account and logged in successfully
Main success scenario	1. Creator asks the System to load a new content 2. System displays the page dedicated to enter new content 3. Creator specify a valid file to be loaded and optionally adds more informations 4. System load the post in Creator's profile and makes it visible to other users
Alternative scenario	3.a Creator specify a non-valid file to be loaded 4.a System shows an error message and returns to MSS 2
Garantee	The database is kept in a consistent state. The new content is visible to each user that accesses Creator's profile

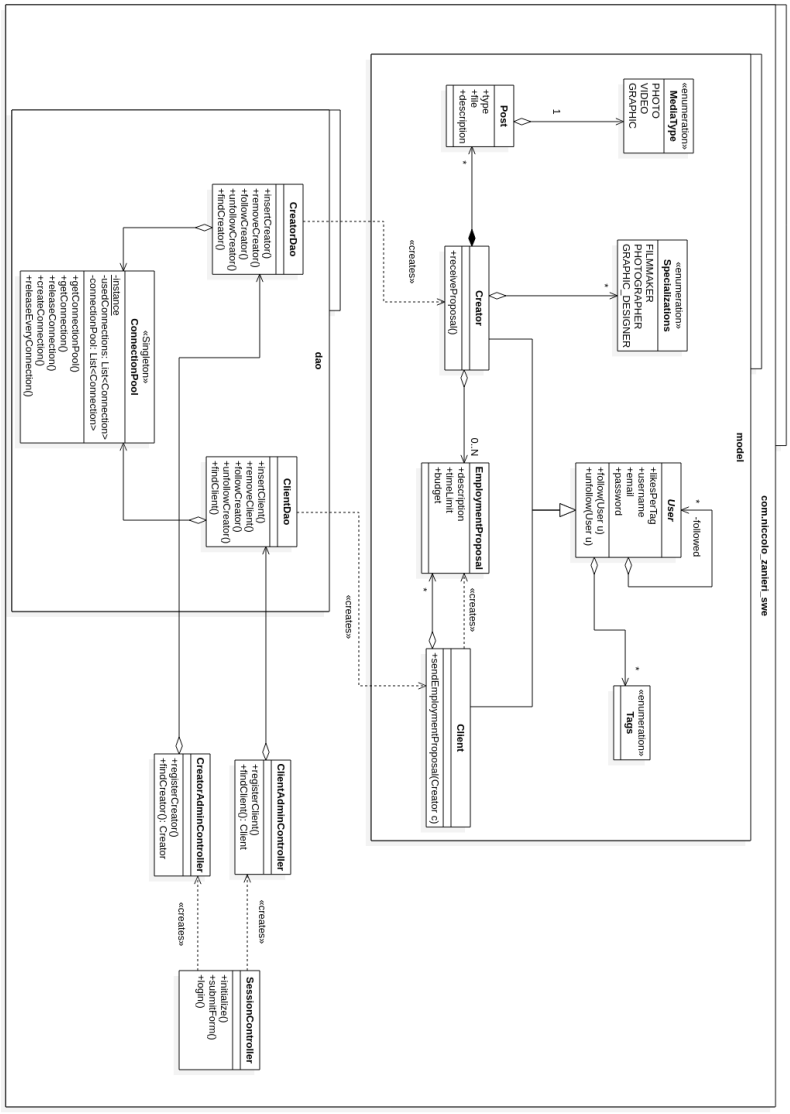
Use-case field	Description
Use-case name	Delete content
Subject area	Basic User Interaction
Actors	Creator
Preconditions	The Creator has a valid account and logged in successfully. Creator is viewing the page of the to-be-deleted content
Main success scenario	1. Creator request to have the content deleted 2. System removes the content from Creator's profile
Garantee	The database is kept in a consistent state. The deleted content is not visible in the app anymore, neither by the Creator nor by other users.

B.2 Creator-Client Interaction

Use-case field	Description
Use-case name	Send employment proposal
Subject area	Creator - Client Interaction
Actors	Creator, Client
Preconditions	Both Creator and Client have a valid account. Client logged in successfully and has access to Creator's profile.
Main success scenario	1. Client file the proposal and requests it to be sent to Creator 2. System add the filed request to Creator active requests
Alternative scenario	2.a Creator has already the maximum number of active proposals he can handle 3.a System show a message to Client saying that the requested couldn't be processed and to try again later
Garantee	No Creator has more active requests than he stated he can handle.

Use-case field	Description
Use-case name	Answer employment proposal
Subject area	Creator – Client Interaction
Actors	Creator, Client
Preconditions	Both Creator and Client have a valid account. Creator logged in successfully and has access to the desired proposal.
Main success scenario	1. Creator decide to either accept or reject the proposal 2. System sends a notification to Client with Creator’s answer to the given proposal

C After implementation class diagram



D Relational DBMS Schema

```
Client(PK(username), email, password)

Creator(PK(username), email, password)

Tag(PK(tag_name))

EmploymentProposal(PK(creator_usr, client_usr), description, time_limit, budget) FK(creator_usr) ref Creator
                                                                                 FK(client_usr) ref Client


Post(PK(creator_usr, title), file, media_type) FK(creator_usr) ref Creator

LikedBy(PK(post_creator, post_title, creator_usr)) FK(post_creator, post_title) ref Post(creator_usr, title)

PostTags(PK(post_creator, post_title), tag) FK(post_creator, post_title) ref Post(creator_usr, title)
                                            FK(tag) ref Tag


FavouriteTags(PK(creator_usr, tag)) FK(creator_usr) ref Creator
                                    FK(tag) ref Tag


Specialization(PK(spec_name))

CreatorSpecializations(PK(creator_usr, follower_usr)) FK(creator_usr) ref Creator
                                                      FK(spec_name) ref Specialization


FollowedByCreator(PK(followed_usr, follower_usr)) FK(followed_usr) ref Creator
                                                  FK(follower_usr) ref Creator

FollowedByClient(PK(followed_usr, follower_usr)) FK(followed_usr) ref Creator
                                                 FK(follower_usr) ref Client
```

Figura D.1: Schema finale utilizzato per la base di dati.