

Relazione elaborato di Ingegneria del Software

Applicativo per la gestione di reviews di videogiochi

Autori:

Pandolfini Luca — Galli Federico

A.A. 2024-2025



Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Contents

1	Introduzione	2
2	Requisiti Software	2
2.1	Requisiti Funzionali	2
2.2	Requisiti Non Funzionali	2
2.3	Use Case	3
2.4	Use Case Template	4
2.5	App Pages	6
3	Progettazione e Implementazione	8
3.1	Class Diagram	8
3.2	Scelte Implementative e Pattern	8
3.2.1	Model-View-Controller	9
3.2.2	Services	10
3.2.3	Dependency Injection	11
3.2.4	Mapper	13
4	Tests	16
4.1	Authorization Test	16
4.2	Reviews Test	16
4.2.1	Post Review Test	16
4.2.2	Double Post Review Test	17
5	Conclusioni	18
5.1	Possibili Aggiunte	18
5.1.1	Sistema di Follow	18
5.1.2	Sistena di Notifiche	18
5.1.3	Sistema di Suggerimenti	18

1 Introduzione

Il seguente elaborato è stato svolto per il conseguimento dell'esame del corso di Ingegneria del Software del terzo anno di ingegneria informatica.

L'idea del progetto è scaturita dalla passione di entrambi per i videogiochi. La nostra volontà era quella di creare un catalogo apposito per videogiochi, che desse la possibilità alle persone della community di fornire pareri personali sui giochi e allo stesso tempo di lasciarsi ispirare per l'acquisto di nuovi titoli dell'ultimo momento. Abbiamo preso come spunto alcune applicazioni già esistenti specifiche nell'ambito di film, serie TV e libri.

La caratteristica principale è la recensione di videogiochi sia attraverso l'utilizzo di un voto sia attraverso l'inserimento di un commento. Ogni videogioco ha le informazioni legate all'anno di rilascio, al produttore, alle console per le quali è disponibile e così via inoltre la cosa più importante è che è corredato di indice di apprezzamento calcolato in base alle votazioni degli utenti.

2 Requisiti Software

2.1 Requisiti Funzionali

L'applicativo si presenta con sola funzione di client. L'utente avrà la possibilità di:

- Registrarsi
- Eseguire il login
- Modificare il proprio profilo
- Ricercare videogiochi per titolo
- Creare una propria wishlist
- Aggiungere e Rimuovere un videogioco dalla wishlist
- Aggiungere una recensione al videogioco
- Aggiungere like alle recensioni dei videogiochi

2.2 Requisiti Non Funzionali

- **Sicurezza:** Protezione dei dati sensibili. Il sistema salva nel database i dati sensibili dell'utente in modo criptato.

2.3 Use Case

Dalla descrizione del dominio del nostro progetto abbiamo identificato gli attori e le loro possibili azioni integrate all'interno dell'applicativo. Abbiamo sviluppato un diagramma di *Use Case* per una spiegazione del funzionamento in toto dell'applicazione più leggibile e di facile comprensione.

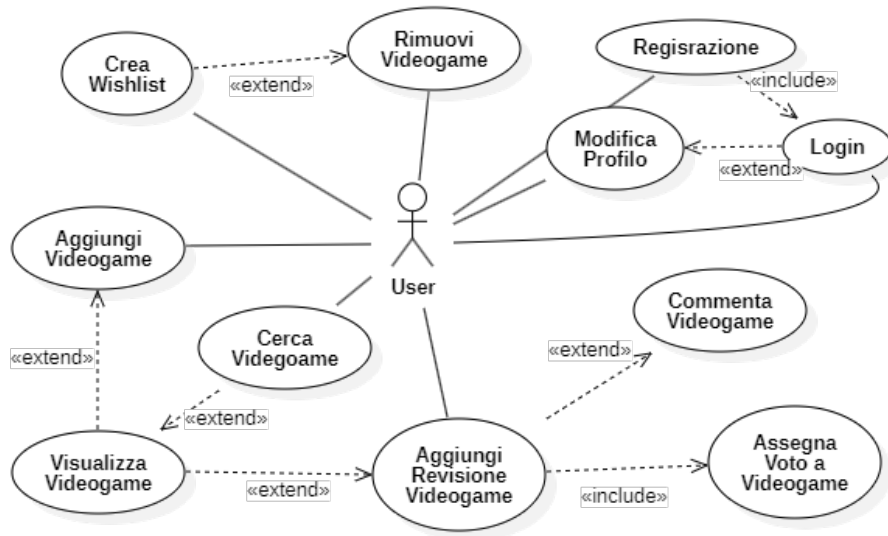


Figure 1: Use Case Diagram per MyVg

2.4 Use Case Template

UseCase 1	Pubblicazione Review
Livello	User Goal
Descrizione	L'utente aggiunge una review pubblica, di un videogioco.
Attori	Utente
Pre-condizioni	L'utente deve essersi autenticato attraverso username e password.
Post-condizioni	La review viene pubblicata sulla pagina del videogioco.
Normale Svolgimento	<ol style="list-style-type: none">1. Inserisce Username e Password per autenticarsi2. Esegue una ricerca di un videogioco3. Seleziona il gioco che vuole recensire dal risultato della ricerca4. Clicca sul bottone "Add Review"5. Sceglie il rating che vuole pubblicare tramite lo spinner6. (opzionale) Scrive un Commento.7. Clicca sul bottone "Post Review"

Table 1: Use Case Template per la pubblicazione

UseCase 2	Aggiunta di un Videogame
Livello	User Goal
Descrizione	L'utente aggiunge un videogame alla sua wishlist
Attori	Utente
Pre-condizioni	L'utente deve essersi autenticato attraverso username e password e aver ricercato il gioco da aggiungere alla wishlist.
Post-condizioni	La wishlist dell'utente viene aggiornata e il gioco aggiunto potrà essere visualizzato nella schermata della wishlist.
Normale Svolgimento	<ol style="list-style-type: none"> 1. Effettua l'accesso attraverso username e password 2. L'utente fa una ricerca tra i giochi. 3. L'utente sceglie uno dei giochi prodotti dalla ricerca aprendo la pagina del gioco 4. L'utente clicca sul bottone <i>"Add to Wishlist"</i>

Table 2: Use Case Template per l'Aggiunta di un Gioco alla Wishlist

2.5 App Pages

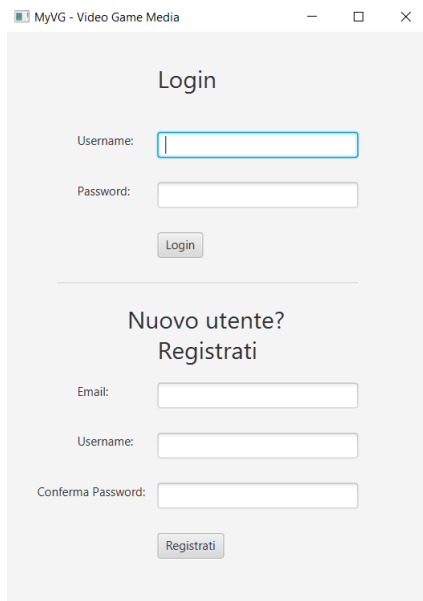


Figure 2: Authentication

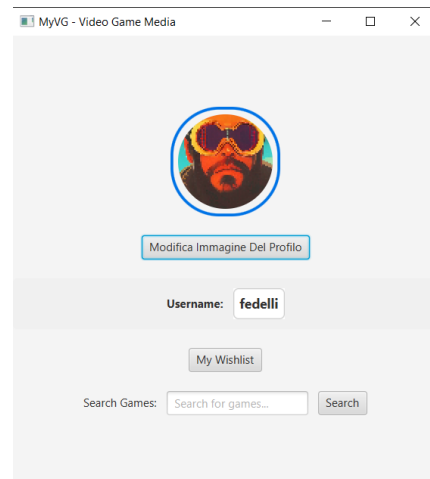


Figure 3: UserProfile Page

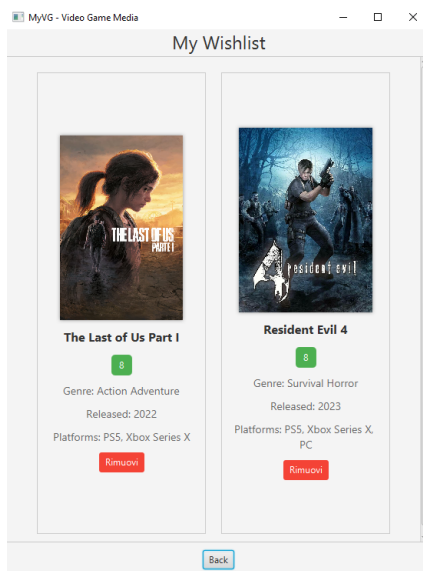


Figure 4: WishlistPage

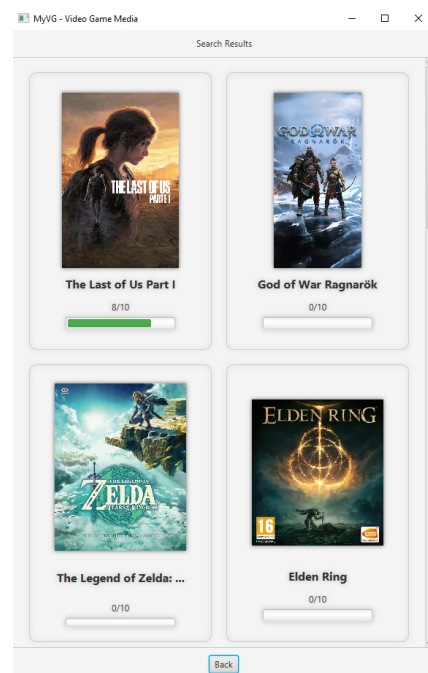


Figure 5: GameSearch Page



Figure 6: Videogame Page

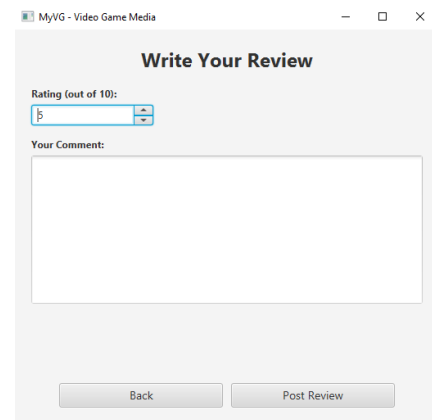


Figure 7: Review Page

3 Progettazione e Implementazione

3.1 Class Diagram

Per sviluppare il nostro progetto ci siamo serviti del linguaggio Java, nello specifico la versione 17 e dell'IDE Visual Studio Code. Per il versionamento del codice abbiamo utilizzato GitHub. Abbiamo sfruttato il framework Springboot integrando la parte grafica con JavaFx, versione 21 e per la gestione dei pacchetti abbiamo utilizzato Maven.

Di seguito riportiamo il Class Diagram che descrive la logica dell'applicazione. Abbiamo mostrato principalmente come comunicano tra di loro i vari livelli e qual è il flusso delle informazioni senza scendere nel dettaglio classe per classe, lasciamo alla parte della descrizione del codice questo compito.

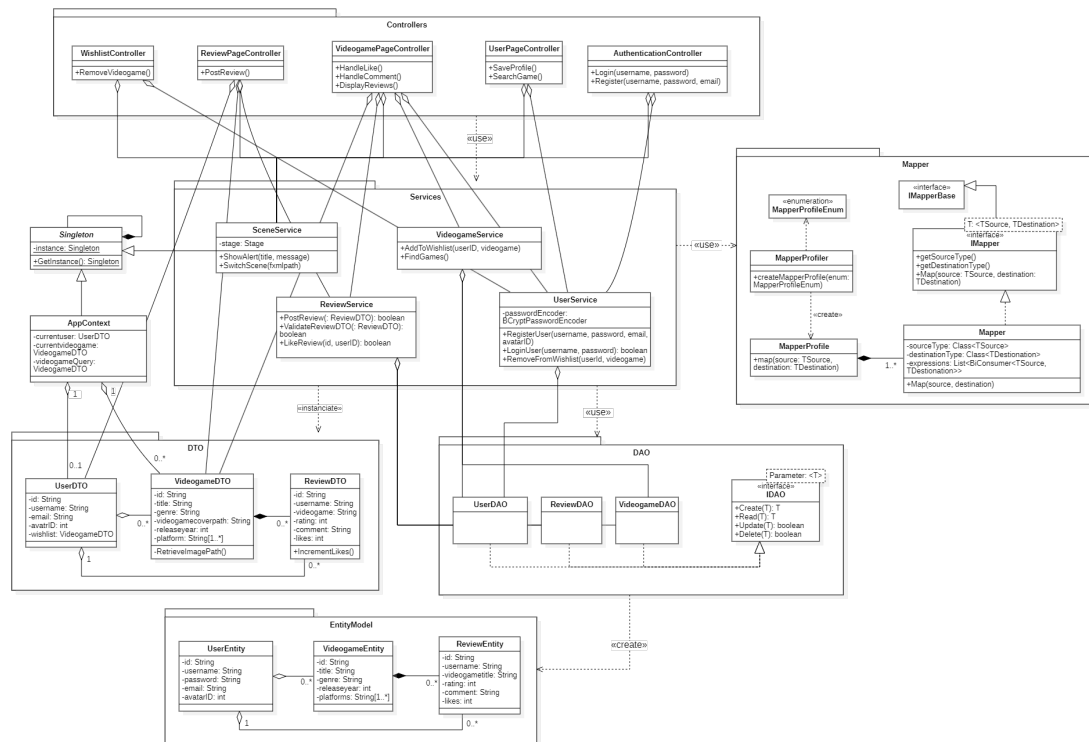


Figure 8: Class Diagram per MyVG

3.2 Scelte Implementative e Pattern

Nella progettazione e implementazione di MyVG, sono state fatte scelte fondamentali riguardanti le tecnologie e i pattern architetturali. Per garantire la robustezza

e la scalabilità del sistema, è stato utilizzato Java Spring Boot come framework di sviluppo per il back-end, permettendo una facile gestione delle dipendenze. L'interfaccia utente è stata sviluppata con JavaFX, per la capacità di creare interfacce grafiche interattive. Per la gestione dei dati, MongoDB è stato scelto come database NoSQL.

1. *Model-View-Controller*
2. *Data Transfer Object*
3. *Mapper*
4. *Dependency Injection*
5. *Data Access Object*

Il nostro progetto utilizza il pattern *Model-View-Controller* per la separazione dei livelli logici e la gestione dell'Interfaccia Grafica. Attraverso l'utilizzo dei pacchetti di JavaFX abbiamo realizzato la parte di UI nonché la *View* del pattern *MVC*.

3.2.1 Model-View-Controller

Abbiamo utilizzato questo pattern per separare l'interfaccia grafica dai dati del modello. L'MVC è, come lascia intendere il nome, diviso in 3 parti:

La parte di *View* viene implementata attraverso JavaFX e l'utilizzo di file *.fxml* che danno la struttura alle pagine attraverso cui l'utente può navigare e interagire. Non inseriremo la parte di codice relativa alla *View* poiché si tratta più che altro di scelte stilistiche e non implementative.

I dati mostrati nelle view relativi al modello della nostra app vengono trasferiti attraverso i **DTO**.

Le interazioni sulla View da parte dell'utente viene gestita dai **Controller**.

Login Controller

Il metodo `HandleLogin()` della classe `AuthenticationController` è utilizzato per eseguire il login di un utente già registrato. Dopo il controllo per l'esistenza dello User nel database e la corrispondenza di Username e Password il metodo fa uso di `SceneService` per cambiare scena e andare alla pagina del profilo dell'utente che ha effettuato il login. `SceneService` è iniettata tramite `Dependency Injection` ed è una classe Singleton

```

@FXML
public void handleLogin() {
    String username = usernameField.getText();
    String password = passwordField.getText();

    if(login(username, password) == true){
        sceneService.switchScene(fxmlPath:"/fxml/UserPage.fxml",
            (UserPageController controller) ->
            {
                controller.setUser();
            });
    }
}

public boolean login(String username, String password) {
    if (userService.loginUser(username, password))
    {
        UserDTO user = userService.getUserByUsername(username);
        AppContext.getInstance().setCurrentUser(user);
        return true;
    }
    else
    {
        sceneService.showAlert(title:"Login Failed", message:"Invalid username or password.");
        return false;
    }
}
}

```

Figure 9: Login Controller

I Controller utilizzano dei **Service** singleton per la business logic. Ad esempio Login Controller utilizza il servizio **SceneService**.

3.2.2 Services

I services vengono usati per rendere la struttura del codice più flessibile e modulare. All'interno del nostro progetto hanno sia ruoli funzionali sia di connessione con il database. I services infatti si servono dei **DAO** e del **Mapper** per il passaggio di dati dal *Database* fino alle *Views* tramite **DTO**, inoltre vengono utilizzati, ad esempio dai *Controllers*, attraverso **Dependency Injection**.

DTO

Il *Data Transfer Object* (DTO) è utilizzato per mantenere data hiding tra i livelli dell'applicazione così da renderla più sicura. Il pattern si occupa appunto di contenere le informazioni necessarie a essere passate dai livelli più bassi a quelli più alti, fino alla UI. Le classi DTO vengono gestite dai **Services** e presentate ai *Controller*.

Scene Service

La classe SceneService è un **Singleton** che fa uso dell'ApplicationContext di Springboot, che noi abbiamo inizializzato nella classe MyVgApplication, per impostare il giusto controller di JavaFx. Quest'ultima ha una sua gestione dei controller per le pagine fxml ma

abbiamo preferito implementare questo metodo così da chiamare delle callback sui vari controller prima che questi vengano mostrati.

```
public <T> void switchScene(String fxmlPath, Consumer<T> initializer) {
    try
    {
        FXMLLoader loader = new FXMLLoader(getClass().getResource(fxmlPath));
        loader.setControllerFactory(applicationContext::getBean);
        Parent root = loader.load();

        stage.setScene(new Scene(root));

        var controller = loader.getController();

        if(initializer != null && controller != null){
            initializer.accept((T)controller);
        }

        stage.show();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

Figure 10: SwitchScene Method

3.2.3 Dependency Injection

La *Dependency Injection* è un design pattern che consente di iniettare le dipendenze di un oggetto dall'esterno anziché crearle all'interno. Lo scopo della *DI* è quello appunto di invertire il controllo degli oggetti rendendoli indipendenti dalle classi in cui vengono utilizzati.

Ci siamo serviti di SpringBoot per l'implementazione di questo design pattern. Nel nostro progetto le classi di tipo Controller sfruttano la *DI* per utilizzare i *Services* di cui hanno bisogno.

Dependency Injection in un Service Attraverso l'evento del click sul bottone di "aggiungi videogame alla wishlist" il *VideogameController* gestisce la richiesta servendosi dell'ApplicationContext per riconoscere l'utente. La classe *VideogameService* è un service che utilizza i servizi di due classi DAO e il Mapper per aggiungere i videogames alle wishlist degli utenti.

```

@FXML
private void onAddToWishlist() {
    //Retrieve context
    UserDTO currentUser = AppContext.getInstance().getCurrentUser();
    VideogameDTO currentGame = AppContext.getInstance().getCurrentVideogame();

    //Check if game already in wishlist
    if(currentUser.getWishlist().stream()
        .anyMatch(game -> game.getTitle().equals(currentGame.getTitle()))) {
        sceneService.showAlert(title:"Already in Wishlist", message:"This game is already in your wishlist!");
        return;
    }

    //Add to wishlist
    videogameService.addToWishlist(currentUser.getId(), currentGame);

    //Update context
    UserDTO user = userService.getUserById(currentUser.getId());
    AppContext.getInstance().setCurrentUser(user);
}

```

Figure 11: AddToWishlist

```

@Service
public class VideogameService {

    private final VideogameDAO videogameDAO;
    private final UserDAO userDAO;

    private final MapperProfile mapperProfile;

    public VideogameService(VideogameDAO videogameDAO, UserDAO userDAO) {
        this.videogameDAO = videogameDAO;
        this.userDAO = userDAO;

        this.mapperProfile = MapperProfileFactory.createMapperProfile(MapperProfileEnum.VIDEOGAME);
    }

    //region CRUD...

    public void addToWishlist(String userId, VideogameDTO videogameDTO) {
        userDAO.read(userId).ifPresent(user -> {
            VideogameEntity videogameEntity = mapperProfile.map(videogameDTO, new VideogameEntity());
            if (!user.getWishlist().contains(videogameEntity)) {
                user.getWishlist().add(videogameEntity);
                userDAO.update(user);
            }
        });
    }

}

```

Figure 12: VideogameService Class

Data Access Object

Il *Data Access Object* (DAO) fornisce un'interfaccia astratta per accedere ai dati da una sorgente (es. database). Abbiamo utilizzato questo pattern per implementare le funzioni di CRUD e creare di conseguenza il nostro modello.

DAO interface Qui viene riportata l'interfaccia comune a tutte le istanze di DAO che implementano il collegamento al database tramite le funzioni CRUD

```
public interface IDAO<T> {  
    //CRUD OPERATIONS  
    Optional<T> create(T t);  
    Optional<T> read(String id);  
    boolean update(T t);  
    boolean delete(String id);  
  
    //UTILS  
    List<T> readAll();  
}
```

Figure 13: DAO Interface

3.2.4 Mapper

Il *Mapper* si occupa di mappare i dati tra oggetti di diverso tipo, spesso tra oggetti di dominio e oggetti DTO. Nel nostro progetto abbiamo implementato un *Mapper* di tipo generico così da utilizzare una mappatura diversa in base alle necessità. *vedi sezione di snippets*

Realizzazione della Mapper Interface

La classe Mapper classe implementa l'interfaccia IMapper. I metodi che definisco la conversione da *TSource* a *TDestination* vengono prima salvati dentro *expressions* e poi attraverso la funzione *map()* vengono eseguiti così da modificare l'istanza di tipo *TDestination* correttamente. Abbiamo lasciato la creazione delle istanze delle classi utilizzate ai metodi esterni che fanno uso di Mapper per mantenere il principio di single responsibility.

```
public interface IMapper<TSource, TDestination> extends IMapperBase {  
    Class<TSource> getSourceType();  
    Class<TDestination> getDestinationType();  
    TDestination map(TSource source, TDestination destination);  
}
```

Figure 14: Mapper Interface

```

public class Mapper<TSource, TDestination> implements IMapper<TSource, TDestination>
{
    private final Class<TSource> sourceType;
    private final Class<TDestination> destinationType;
    private List<BiConsumer<TSource, TDestination>> expressions = new ArrayList<>();
    public Mapper(Class<TSource> sourceType, Class<TDestination> destinationType) {
        this.sourceType = sourceType;
        this.destinationType = destinationType;
    }
    public List<BiConsumer<TSource, TDestination>> getExpressions(){return expressions;}
    public void setExpressions(List<BiConsumer<TSource, TDestination>> expressions){
        this.expressions = expressions;
    }
    @Override
    public Class<TSource> getSourceType(){return sourceType;}
    @Override
    public Class<TDestination> getDestinationType(){return destinationType;}
    @Override
    public TDestination map(TSource source, TDestination destination){
        for(BiConsumer<TSource, TDestination> exp : expressions)
        {
            exp.accept(source, destination);
        }
        return destination;
    }
}

```

Figure 15: Mapper Class

Creazione dei MapperProfile

Attraverso la funzione *CreateMapperProfile()* che sfrutta una Enumeration, vengono creati dei mapper di diverso tipo su richiesta, tipicamente dei *Services*. Questo metodo imposta delle funzioni per ogni configurazione possibile, in questo caso abbiamo mostrato solo quella da *VideogameDTO* a *VideogameEntity* e viceversa.

```

public static MapperProfile createMapperProfile(MapperProfileEnum profile) {
    switch (profile) {
        case VIDEOGAME:
            var mapperReview = MapperProfileFactory.createMapperProfile(MapperProfileEnum.REVIEW);
            return new MapperProfile() {{
                setMappers(List.of(
                    new Mapper<VideogameDTO, VideogameEntity>(VideogameDTO.class, VideogameEntity.class) {{
                        setExpressions(List.of(
                            (source, destination) -> destination.setId(source.getId()),
                            (source, destination) -> destination.setTitle(source.getTitle()),
                            (source, destination) -> destination.setGenre(source.getGenre()),
                            (source, destination) -> destination.setReleaseYear(source.getReleaseYear()),
                            (source, destination) -> destination.setPlatform(source.getPlatform()),
                            (source, destination) -> destination.setReviews(source.getReviews().stream()
                                .map(reviewDTO -> mapperReview.map(reviewDTO, new ReviewEntity()))
                                .collect(Collectors.toList()))
                        ));
                    }),
                    new Mapper<VideogameEntity, VideogameDTO>(VideogameEntity.class, VideogameDTO.class) {{
                        setExpressions(List.of(
                            (source, destination) -> destination.setId(source.getId()),
                            (source, destination) -> destination.setTitle(source.getTitle()),
                            (source, destination) -> destination.setGenre(source.getGenre()),
                            (source, destination) -> destination.setReleaseYear(source.getReleaseYear()),
                            (source, destination) -> destination.setPlatform(source.getPlatform()),
                            (source, destination) -> destination.setReviews(source.getReviews().stream()
                                .map(reviewEntity -> mapperReview.map(reviewEntity, new ReviewDTO()))
                                .collect(Collectors.toList()))
                        ));
                    })
                ));
            }};
    }
}

```

Figure 16: MapperProfile Creation

```

public class MapperProfile{
    private List<IMapper<?, ?>> mappers = new ArrayList<>();

    public <TSource, TDestination> TDestination map(TSource source, TDestination destination)
    {
        for (IMapper<?, ?> mapper : mappers) {
            if (mapper.getSourceType().isAssignableFrom(source.getClass()) &&
                mapper.getDestinationType().isAssignableFrom(destination.getClass())) {
                var mapToUse = (IMapper<TSource, TDestination>) mapper;
                return mapToUse.map(source, destination);
            }
        }

        throw new IllegalArgumentException("Mapper not found for: " +
            source.getClass().getName() + " -> " + destination.getClass().getName());
    }
}

```

Figure 17: Enter Caption

4 Tests

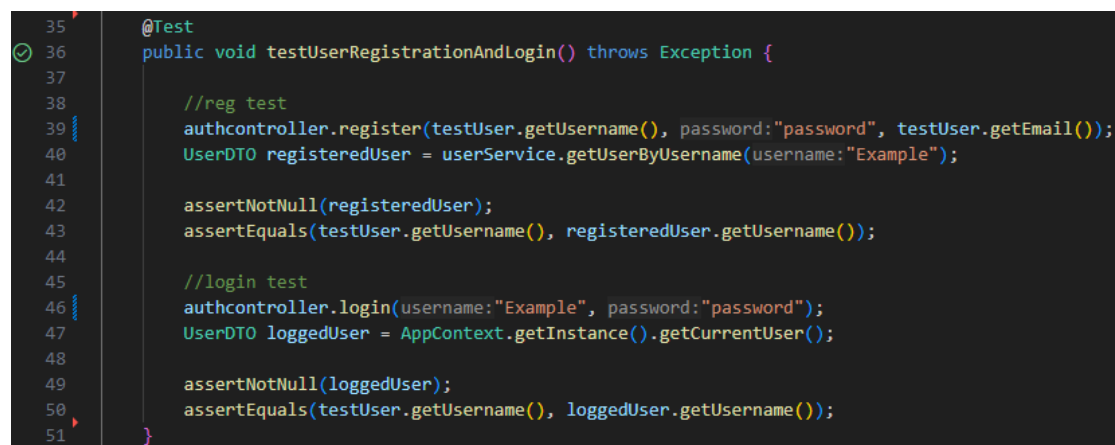
Di seguito riportiamo alcuni test fatti per controllare l'integrazione tra le classi e il corretto funzionamento di singoli metodi.

Abbiamo sfruttato il framework per test di SpringBoot ovvero *SpringbootTest* per l'utilizzo della dependency injection e l'inizializzazione dei test.

4.1 Authorization Test

Abbiamo implementato il seguente test poiché una delle funzionalità principali della nostra applicazione è proprio quella della creazione di un profilo e quella di login di un utente.

Abbiamo controllato i metodi dell'*AuthorizationController* per eseguire una registrazione e un login e abbiamo verificato sia che l'utente creato non esistesse già sia che le credenziali usate per accedere all'app appartenessero ad un utente già registrato e in caso positivo fossero corrette.



```
35
36 @Test
37 public void testUserRegistrationAndLogin() throws Exception {
38
39     //reg test
40     authcontroller.register(testUser.getUsername(), password:"password", testUser.getEmail());
41     UserDTO registeredUser = userService.getUserByUsername(username:"Example");
42
43     assertNotNull(registeredUser);
44     assertEquals(testUser.getUsername(), registeredUser.getUsername());
45
46     //login test
47     authcontroller.login(username:"Example", password:"password");
48     UserDTO loggedUser = AppContext.getInstance().getCurrentUser();
49
50     assertNotNull(loggedUser);
51     assertEquals(testUser.getUsername(), loggedUser.getUsername());
52 }
```

Figure 18: Authentication Test

4.2 Reviews Test

Abbiamo sviluppato i test sulle *Reviews* per garantire l'affidabilità e la correttezza dell'aspetto principale della nostra applicazione, ovvero la possibilità di valutare videogiochi attraverso voti e commenti.

4.2.1 Post Review Test

Questo test è stato scritto per controllare che una review postata presenti dei parametri corretti. In particolare abbiamo testato il caso in cui si cerca di assegnare un voto non

compreso tra 1 e 10 ad un videogioco e il caso in cui il numero di likes è minore di 0.

```
24  @Test
25  void testPostReview() {
26      // Arrange
27      ReviewDTO invalidRating = new ReviewDTO(username:"testUser",
28                                              videogame:"testVideogame", -1, comment:"Great game!", likes:10);
29      ReviewDTO invalidLikes = new ReviewDTO(username:"testUser",
30                                              videogame:"testVideogame", rating:8, comment:"", -1);
31      ReviewDTO validReview = new ReviewDTO(username:"testUser",
32                                              videogame:"The Last of Us Part I", rating:8, comment:"Great game!", likes:10);
33      // Act & Assert
34      assertThrows(IllegalArgumentException.class, () -> {
35          reviewService.postReview(invalidRating);
36      });
37
38      assertThrows(IllegalArgumentException.class, () -> {
39          reviewService.postReview(invalidLikes);
40      });
41      boolean result = reviewService.postReview(validReview);
42      assertEquals(result, true);
43      reviewService.delete(validReview.getId());
44  }
```

Figure 19: PostReview Test

4.2.2 Double Post Review Test

Attraverso questo test abbiamo verificato il corretto funzionamento di post di una review nel caso in cui un utente cerchi di aggiungerne più di una. Abbiamo creato due revisioni compilate dallo stesso utente sullo stesso videogioco così da verificare che il sistema neghi questa possibilità all'utente.

```
46  @Test
47  void testUserCannotReviewSameGameTwice() {
48      // Arrange
49      ReviewDTO firstReview = new ReviewDTO(username:"testUser",
50                                              videogame:"The Last of Us Part I", rating:8, comment:"Great game!", likes:10);
51      ReviewDTO secondReview = new ReviewDTO(
52          username:"testUser", videogame:"The Last of Us Part I", rating:8, comment:"Great game!", likes:10);
53      // Act & Assert
54      reviewService.postReview(firstReview);
55
56      assertThrows(IllegalArgumentException.class, () -> {
57          reviewService.postReview(secondReview);
58          reviewService.delete(secondReview.getId());
59      });
60
61      reviewService.delete(firstReview.getId());
62  }
63
64  }
```

Figure 20: Double Post Review Test

5 Conclusioni

5.1 Possibili Aggiunte

Nel progetto abbiamo inserito solo le funzioni principali e necessarie al funzionamento di un applicativo per come lo abbiamo pensato. Di seguito riportiamo alcune delle possibili features che potrebbero essere aggiunte al fine di rendere l'applicativo una vera e propria applicazione da inserire nel mercato.

- I. Sistema di follow
- II. Sistema di notifiche
- III. Sistema di suggerimenti

5.1.1 Sistema di Follow

L'applicazione potrebbe consentire di seguire altri utenti con la possibilità di accesso a determinati dati non sensibili. Un utente potrebbe visionare la wishlist della persona seguita, vedere quali sono i giochi a cui ha giocato e quanto gli sono piaciuti. In questo modo le persone più seguite e potenzialmente quelle con più incidenza nel mondo dei videogiochi potrebbero consigliare i migliori giochi o i cosiddetti *"must to play"*.

5.1.2 Sistema di Notifiche

Data la presenza di una wishlist potrebbe essere necessario un sistema di notifiche che avvisino quando quel gioco è disponibile in caso non sia ancora uscito o ci siano novità su prezzi. Le notifiche potrebbero andare a coprire anche le funzionalità necessarie legate all'inserimento del sistema di follow.

5.1.3 Sistema di Suggerimenti

Con la possibilità di dare un voto e quindi un apprezzamento a diverse tipologie di giochi ed essendo questi appartenenti a più categorie, l'applicativo potrebbe dare consigli all'utente su nuovi giochi da provare.