UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola
di Ingegneria

Corso di Laurea Triennale
in
Ingegneria Informatica

Elaborato di SWE

Fabio Luccioletti e Gianmarco Pastore

Applicativo Java per la ricerca e la gestione di musei italiani

Anno Accademico 2020-2021

Indice

1 Motivazione e Contenuti

L'elaborato che abbiamo realizzato è un'applicazione software per la gestione e la fruizione di informazioni relative ad un gran numero di musei italiani. Per la sua realizzazione abbiamo sfruttare un **database relazionale** che contiene le informazioni sui vari musei, estratte da Wikipedia tramite un processo di **web scraping**.
Il database è gestito da un *amministratore* che ha la responsabilità di mantenere i dati corretti e aggiornati: riceve le segnalazioni di errori e dopo averne verificato la validità può apportare modifiche al database. Oltre a correggere le informazioni si occupa di aggiungere nuovi musei e rimuovere quelli che non aderiscono più al servizio.
Tutti gli *utenti* che vogliano usufruire dei servizi offerti dal software devono eseguire un login con le rispettive credenziali. L'applicazione fornisce agli utenti la possibilità di ricercare tra i musei presenti nel database attraverso una barra di ricerca, e poi di accedere alle relative informazioni. La funzione di ricerca può essere impostata secondo diversi criteri, che ne determinano i risultati: ad esempio si potrà ricercare per *parole chiave*, per *distanza* da una certa località o per *positività delle recensioni*. Inoltre gli utenti possono anche prenotare visite ad un museo oppure lasciare delle recensioni.
I servizi del software sono rivolti anche ai *proprietari dei musei*, i quali possono richiedere l'aggiunta (ma anche la rimozione) del proprio museo a quelli presenti nel database. Il proprietario di un museo che è presente nel database può accedere ad un'interfaccia che gli permette di eseguire varie operazioni. Può modificare le informazioni del museo, aggiungere eventi (riduzione prezzi dei biglietti, mostre particolari...) o visualizzare una serie di statistiche come il numero di visite alla pagina o il numero di prenotazioni.
L'applicazione include anche un meccanismo di segnalazione degli errori; gli utenti possono effettuare delle segnalazioni se notano inesattezze o informazioni mancanti per un dato museo. Queste vengono esaminate dai proprietari dei rispettivi musei che hanno la possibilità di approvarle o meno. Le richieste approvate vengono allora inviate all'amministratore, che procede alla modifica. Eventuali modifiche possono essere richieste anche direttamente dal proprietario di un museo e in questo caso ovviamente non necessitano di approvazione.

2 Analisi dei requisiti

2.1 Requisiti funzionali

I requisiti funzionali della nostra applicazione sono rappresentati nei seguenti **Use Case Diagram**. Essendo l'applicazione abbastanza complessa abbiamo realizzato due Use Case Diagram, uno di livello *user goal* (figura 1) e uno di livello *summary* (figura 2).

Abbiamo anche documentato dei casi d'uso con degli **Use Case Template**. In figura 3 il caso d'uso relativo alla ricerca di un museo.

In figura 4 un **mock-up** (tratto direttamente dall'interfaccia che abbiamo realizzato con l'ausilio della libreria *Swing*) che rappresenta la pagina in cui sono svolte queste operazioni.

Il meccanismo di report è descritto nello Use Case Template in figura 5, considerato a livello summary, e nei mock-up in figura 6.

2.2 Requisiti strutturali

Il software per la ricerca e gestione dei musei necessita, per il salvataggio e il recupero delle informazioni, di collegarsi ad un database. In particolare per il nostro progetto abbiamo deciso di affidarci ad un DBMS **PostgreSQL**. Il DBMS è fisicamente installato su un *RaspberryPi 4*, accessibile dalla rete esterna grazie ad una regola di *port forwarding* che espone la porta 5432.

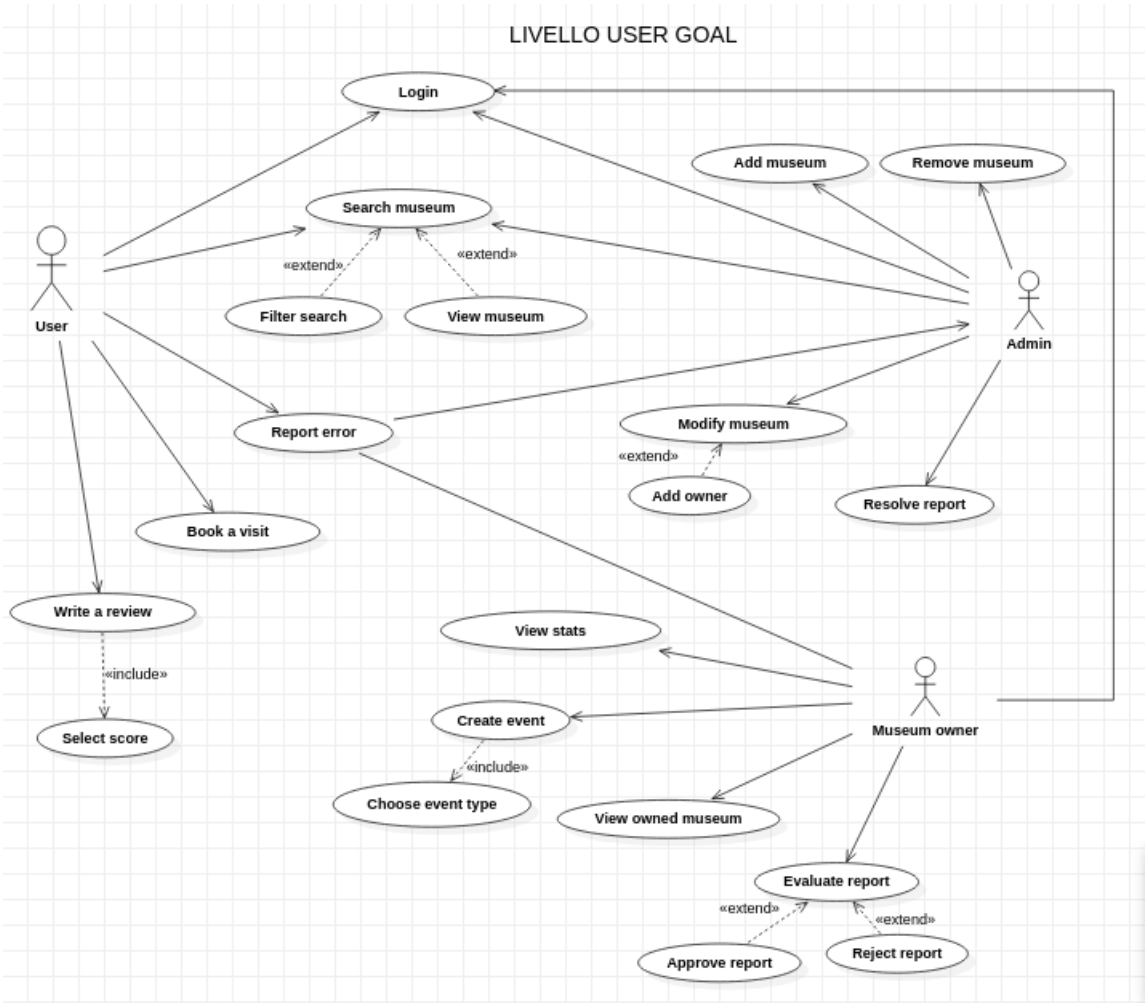

Figura 1: User Case Diagram - livello user goal

Nel database sono presenti 11 tabelle.
Per poter sfruttare due funzionalità indispensabili (la ricerca dei musei per parole chiave e l'autenticazione degli utenti) sono state necessarie delle configurazioni ulteriori. Più specificatamente abbiamo installato l'estensione **pgcrypto**, che fornisce una serie di funzioni per il salvataggio criptato delle password inserite dall'utente e per l'autenticazione. Inoltre abbiamo creato nella tabella dei musei una colonna di tipo *tsvector*, che rappresenta l'indice per le ricerche **Full Text** eseguite. Il tsvector di ogni museo tiene in considerazione per l'indicizzazione il nome e la descrizione di tale museo; è tenuto aggiornato tramite due **trigger**, che si attivano rispettivamente all'aggiunta di un nuovo museo e alla modifica del campo *descrizione* e/o *nome* di un museo esistente.

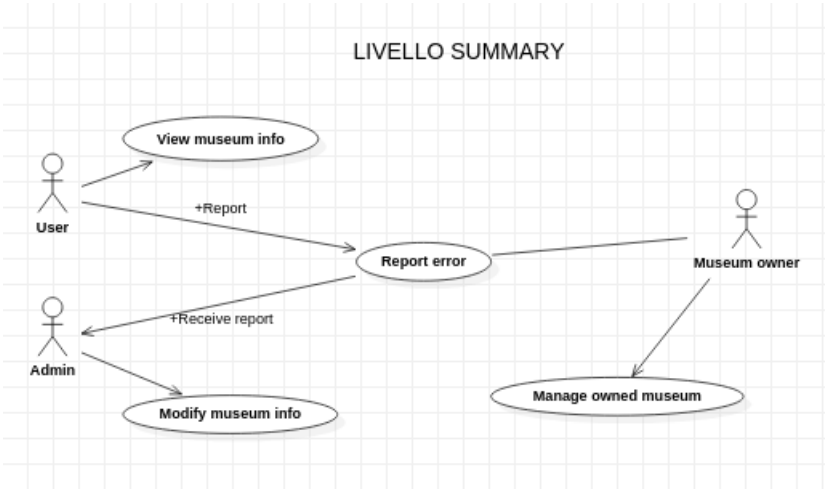

Figura 2: User Case Diagram - livello summary

UC	Search museum
Level	User goal
History	Created 30/12/2020 Gianmarco Pastore & Fabio Luccioletti
Description	L'utente ricerca un museo tra quelli disponibili e ne visualizza le informazioni
Actors	Utente
Pre-conditions	L'utente deve aver fatto il login con il suo account
Post-conditions	Il sistema deve memorizzare le operazioni dell'utente
Basic course	1. Il caso inizia quando l'utente apre la pagina per la ricerca dei musei 2. L'utente inserisce nell'interfaccia i criteri della ricerca e preme il pulsante CERCA 3. Il sistema mostra i risultati della ricerca 4. L'utente seleziona il risultato a cui è interessato 5. Il sistema mostra la pagina iniziale del museo 6. L'utente può navigare tra le pagine di informazioni del museo selezionato
Alternative course	2a. Il sitema non trova risultati per la ricerca 2a1. Il sistema suggerisce di modificare i criteri di ricerca 3a. L'utente non è interessato a nessun risultato 3a1. L'utente torna alla pagina di ricerca 6a. Il sistema non possiede pagina di informazioni del museo 6a1. L'utente può segnalare il problema ad un amministratore
Non-Functional	Architettura: sfrutta database relazionale per recuperare le informazioni dei musei Performance: tempi di attesa delle ricerche $\leq$ 5secondi.

Figura 3: Use Case Template - ricerca di un museo

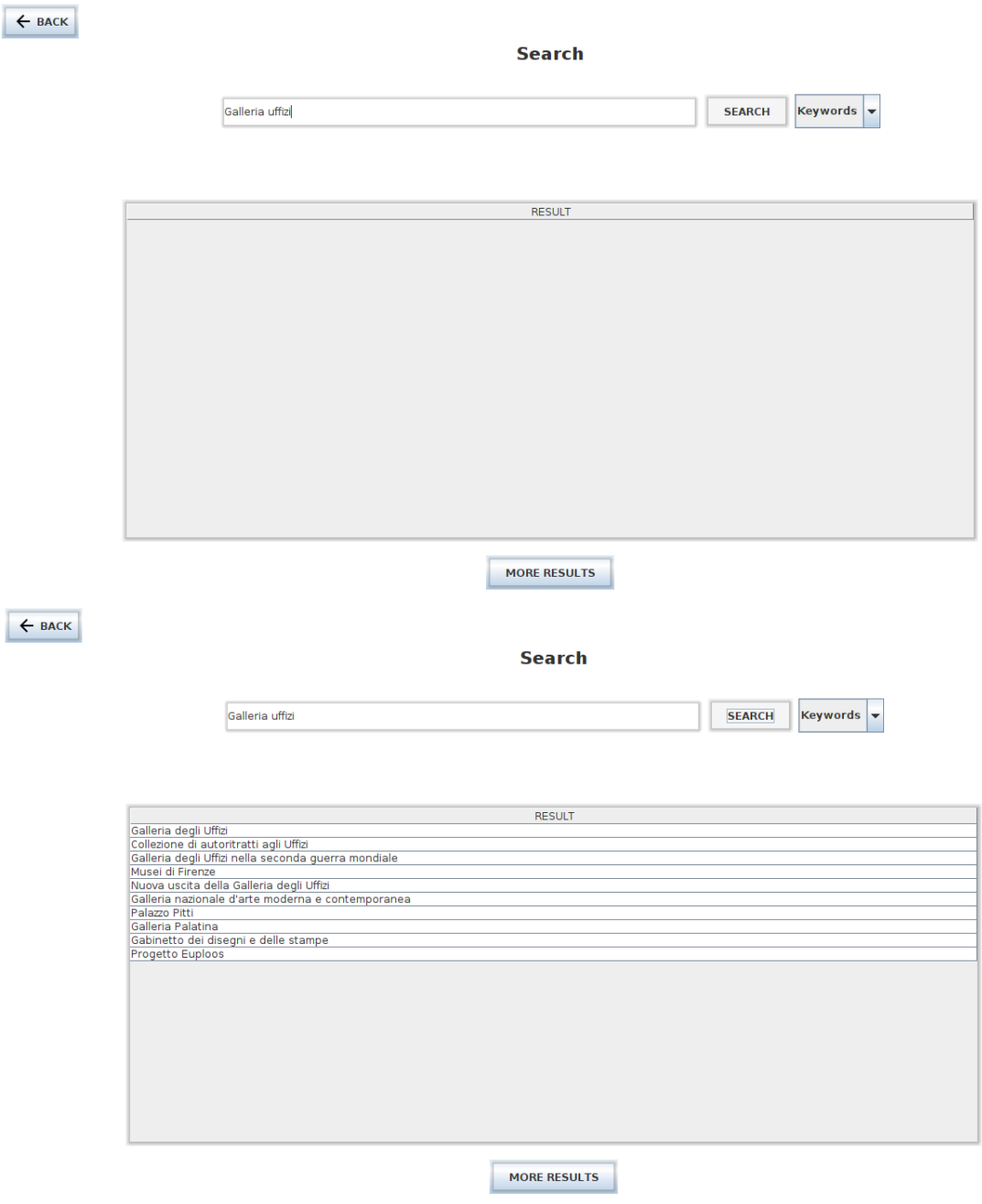

Figura 4: Pagina di ricerca

UC	Report error
Level	Summary goal
History	Created 04/01/2021 Gianmarco Pastore & Fabio Luccioletti
Description	L'utente segnala un errore in un museo, che, se approvato dal rispettivo proprietario, viene inviato all'admin
Actors	User, Admin, Museum owner
Pre-conditions	L'utente deve aver fatto il login con il suo account, e si trova sulla pagina di un museo
Post-conditions	Il sistema deve memorizzare le eventuali modifiche dell'admin
Basic course	1. Il caso inizia quando l'utente preme il tasto report nella pagina del museo 2. Il sistema mostra un campo in cui inserire la segnalazione 3. L'utente scrive la segnalazione e preme il tasto per la conferma 4. Il sistema manda la segnalazione al proprietario del museo coinvolto dalla segnalazione 5. Il proprietario del museo approva la segnalazione cliccando sul bottone apposito 6. Il sistema manda la segnalazione all'admin 7. L'admin effettua le opportune modifiche e preme il tasto di conferma 8. Il sistema elimina la segnalazione dal database
Alternative course	1a. Il caso inizia quando il proprietario del museo preme il tasto report nella pagina del suo museo. 1a1. Il sistema mostra un campo in cui inserire la segnalazione 1a2. Il proprietario scrive la segnalazione e preme il tasto per la conferma 1a3. continua come dal punto 6. 5a. Il proprietario del museo non approva la segnalazione 5a1. Il sistema elimina la segnalazione dal database.
Non-Functional	Architettura: sfrutta database relazionale per memorizzare le segnalazioni

Figura 5: Use Case Template - segnalazione di un errore

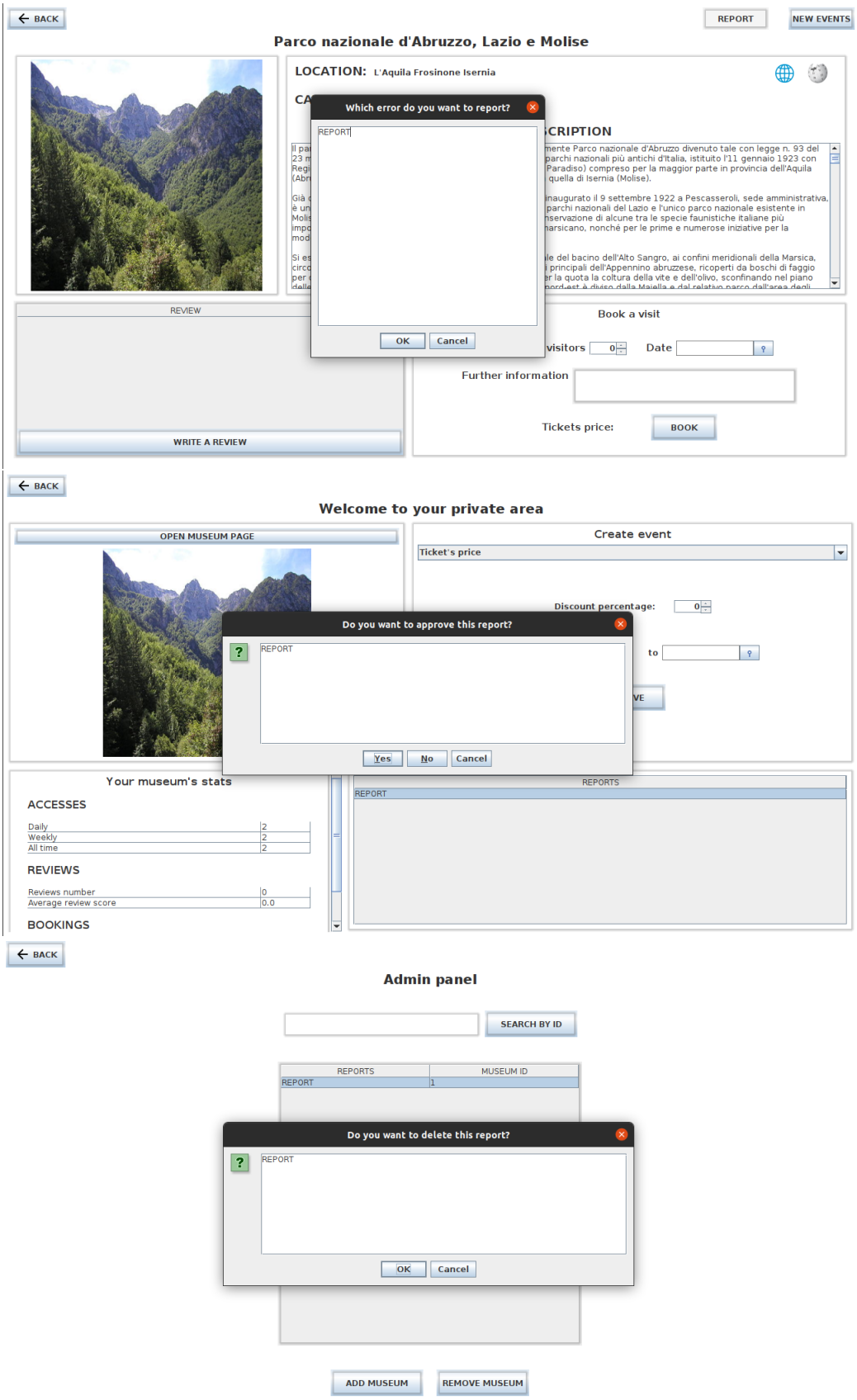

Figura 6: Pagine coinvolte nei report

3 Progettazione

In Figura 8 è rappresentato il **class diagram** dell'applicazione, che è stato aggiornato successivamente alla scrittura del codice.

Inoltre per la realizzazione della nostra applicazione abbiamo bisogno di alcuni specifici **design pattern** (paragrafo 4.2) come **MVC** e **Gateway** che hanno comportato una divisione del codice in diversi *package*. Inoltre la presenza di una *GUI* (implementata tramite java *Swing* e *Swingx*) e di un database da cui estrarre i dati, implicano una stratificazione ulteriore dell'applicazione. Per rappresentare tutte queste queste relazioni abbiamo realizzato un **model diagram** Figura 7.

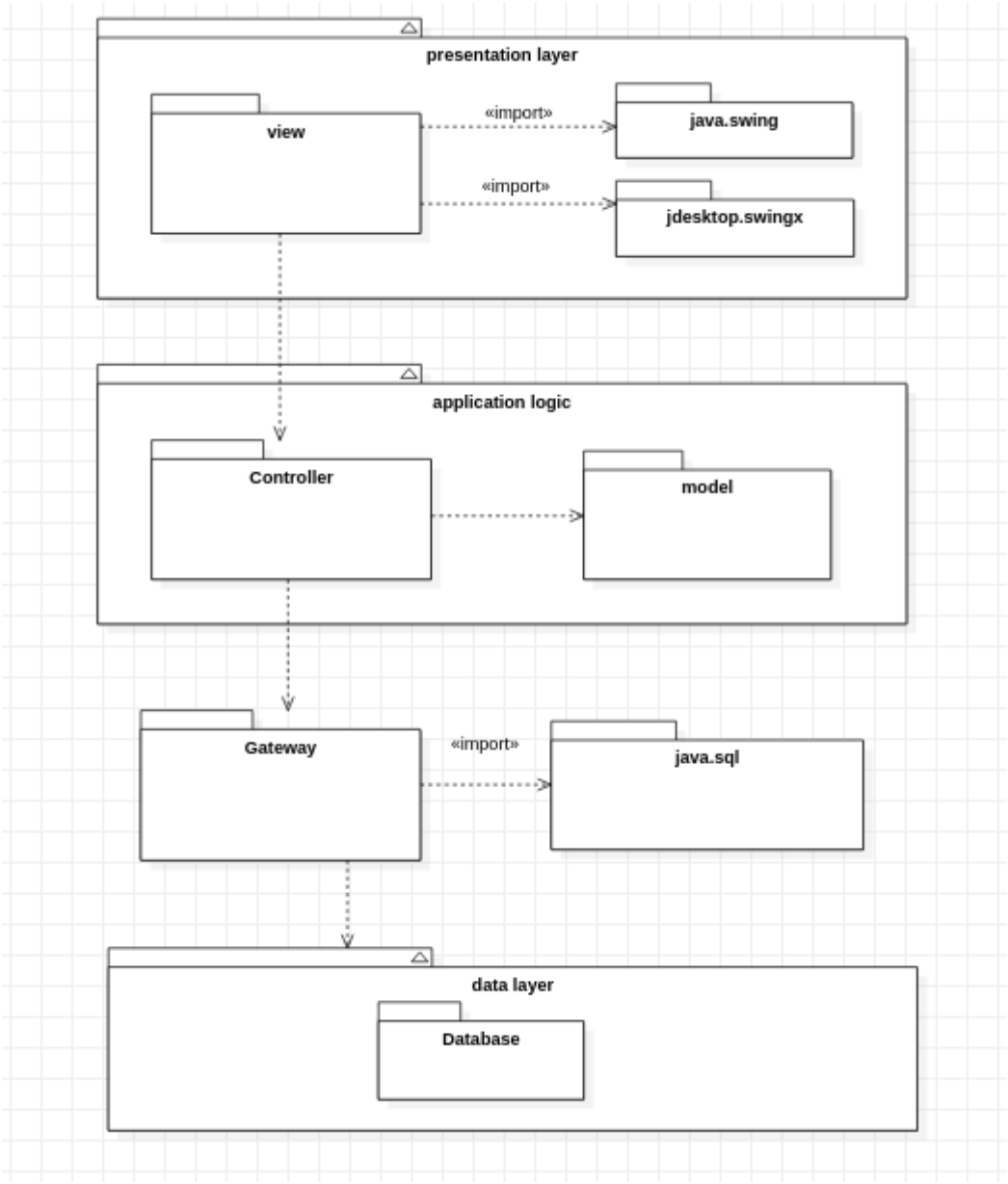

Figura 7: Model Diagram

Figura 8: Class Diagram

4 Implementazione

La nostra applicazione consiste di quattro *package* (più il *main*): **model**, **view**, **controller**, **gateway**.

4.1 Classi

In questa sezione vengono descritte le classi che fanno parte dei package model e gateway.

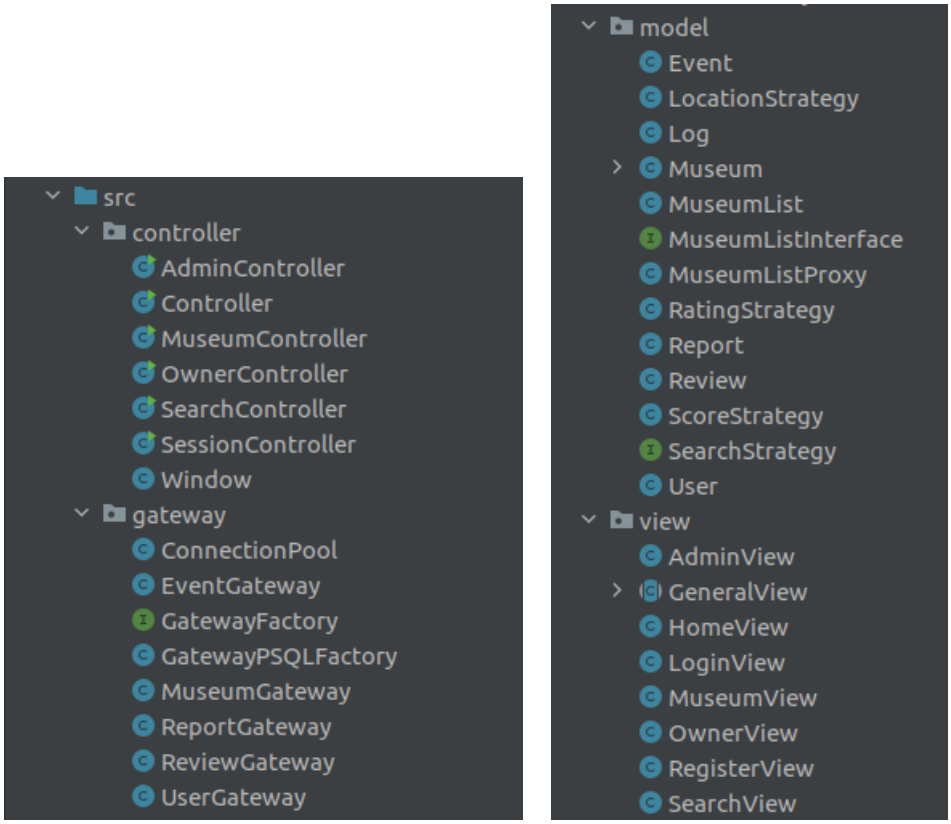

Figura 9: Suddivisione in package

4.1.1 GatewayFactory

Interface **GatewayFactory** GatewayPSQLFactory	
• Provides methods in order to create all types of gateways	

4.1.2 GatewayPSQLGateway

GatewayPSQLFactory	GatewayFactory
• Provides instances of the gateways that uses postgreSQL queries	• UserGateway • MuseumGateway • EventGateway • ReviewGateway • ReportGateway

4.1.3 ConnectionPool

ConnectionPool	
• Creates and hold a predetermined number of connections • Provides connections to gateways that require it • Close the connections when the program ends	• UserGateway • MuseumGateway • ReviewGateway • ReportGateway • EventGateway

4.1.4 UserGateway

UserGateway	
• Interact with the database in order to provide requested data • Manages login and registration of users • Stores and removes users reports • Registers user accesses to museums	• SessionController • AdminController • OwnerController • Log • User

4.1.5 MuseumGateway

MuseumGateway	
• Searches for museums in the database according to a certain strategy • Adds, removes and modifies museums from the database • Stores and retrieves informations about bookings	• AdminController • OwnerController • SearchController • MuseumController

4.1.6 ReportGateway

ReportGateway	
• Stores and retrieves reports from database • Manages the approval of reports	• AdminController • MuseumController • OwnerController

4.1.7 ReviewGateway

ReviewGateway	
• Stores and retrieves reviews from database • Retrieves stats about museum's reviews	• MuseumController • OwnerController

4.1.8 EventGateway

EventGateway	
• Stores and retrieves events from database • Retrieves informations linked to events (like museum's closure or price reduction)	• OwnerController • MuseumController

4.1.9 Log

Log	
• Sets up and provide a logger to the other classes • Given an Exception, returns its stack trace as a String	• All the other classes

4.1.10 MuseumListInterface

Interface MuseumListInterface MuseumList, MuseumListProxy	
• Provides methods for the subclasses	• SearchController

4.1.11 MuseumList

MuseumList MuseumListInterface	
• Keeps the list of museums (stored in a JSONArray) obtained from the research • Creates the number of museums requested	• MuseumListProxy • Museum

4.1.12 MuseumListProxy

MuseumListProxy MuseumListInterface	
• Implements lazy initialization for the museum list • Implements caching for the museum list	• MuseumList • Museum

4.1.13 SearchStrategy

Interface SearchStrategy ScoreStrategy, LocationStrategy, RatingStrategy	
• Defines the method used to build an SQL select query	

4.1.14 LocationStrategy

LocationStrategy	SearchStrategy
• Provides a SQL query which can be used to search for museums ordered by distance from a given location	• MuseumGateway

4.1.15 ScoreStrategy

ScoreStrategy	SearchStrategy
• Provides a SQL query which can be used to search for museums ordered only by relevance (relatively to the keywords)	• MuseumGateway

4.1.16 RatingStrategy

RatingStrategy	SearchStartegy
• Provides a SQL query which can be used to search for museums ordered by rating	• MuseumGateway

4.1.17 User

User	
• Represents a user, its attributes map the columns of the respective table in the database	• UserGateway • Window • Controller

4.1.18 Museum

Museum	
• Represents a museum, its attributes map the columns of the respective table in the database	• MuseumGateway • MuseumListInterface • Controller

4.1.19 Report

Report	
• Represents a report, its attributes map the columns of the respective table in the database	• ReportGateway • Controller

4.1.20 Event

Event	
• Represents an event, its attributes map the columns of the respective table in the database	• EventGateway • Controller • Museum

4.1.21 Review

Review	
• Represents a review, its attributes map the columns of the respective table in the database	• ReviewGateway • Controller • Museum

4.2 Design Pattern

Nel nostro applicativo abbiamo utilizzato i seguenti **Design Pattern**.

- **Gateway** (DAO)
  Il dialogo con una base dati è un aspetto indispensabile nel nostro progetto; i *controller* e le altri classi facenti parte dell'**application logic** hanno spesso necessità di ricevere specifiche informazioni dal **data layer**. Per nascondere la complessità di quest'ultimo

al resto dell'applicazione abbiamo deciso di usare il *pattern strutturale* **Data Access Object** (o **Table Data Gateway**, nel lessico di *Martin Fowler*). Ognuno dei 5 Gateway incapsula una tabella del database (talvolta più di una se il loro contesto di utilizzo è il medesimo) e contiene nei suoi metodi tutto il codice *SQL* necessario all'interrogazione.

- **Object pool**
  Un'altra necessità dovuta al dialogo con il database, è stata quella di mantenere con esso una *connessione* affidabile e sempre disponibile. Aprire una nuova connessione ogni volta ce ne fosse stata la necessità era un'operazione troppo onerosa; d'altra parte mantenere aperta la stessa connessione, condividendola tra le varie classi *Gateway* non era una buona idea.
  Il compito della classe **ConnectionPool** è appunto quello di creare, gestire e, eventualmente, chiudere le connessioni al database. Il programma all'avvio chiede al ConnectionPool di aprire 5 connessioni; ogni volta che un Gateway deve eseguire una query, ne richiede una e dopo il suo utilizzo la restituisce.
  Il ConnectionPool, prima di fornire una connessione, testa se è ancora valida e, se non lo è, la sostituisce. Al termine dell'esecuzione tutte le connessioni ancora aperte vengono chiuse.

- **Singleton**
  Per alcune classi ci è sembrato appropriato usare il pattern *Singleton*. Più specificatamente, si è reso necessario quando dovevamo controllare l'accesso ad una **risorsa condivisa,** nel nostro caso un *file* ed il *database*. La prima casistica è quella della classe **Log**, che inizializza un *logger* con le impostazioni appropriate e lo rende disponibile nel resto del codice: l'utilizzo di un Singleton è ancor più legittimato dato che permette in modo sicuro di ottenere l'istanza dell'oggetto di interesse praticamente **ovunque** (e del resto il *logging* è, al giusto livello, sempre appropriato dopo ogni azione). Un discorso analogo può essere fatto per i **Gateway**, che si relazionano con il database e la cui unica istanza viene fornita a varie classi attraverso l'utilizzo di una *Factory*.

- **Proxy**
  Per memorizzare e operare con la lista di musei che costituisce il risultato di una ricerca abbiamo creato una classe apposita **MuseumList**. Gli oggetti di questo tipo prendono come parametro nel loro costruttore la *query* relativa alla ricerca dell'utente e creano così la lista dei risultati. Dato che questa lista può essere molto lunga e viene creata contemporaneamente all'oggetto che la contiene, abbiamo deciso di utilizzare il pattern **Proxy** per implementare la **lazy initialization**. Tramite il proxy, infatti, creiamo la lista dei risultati solo quando questa è realmente richiesta, ovvero nel momento in cui viene effettuata una ricerca. Oltre a gestire il *lifecycle* degli oggetti di tipo MuseumList, questo proxy ci permette di fare **caching**, restituendo la stessa istanza della lista nel caso l'utente richieda la visualizzazione di più risultati.

- **Strategy**
  L'utente può voler ricercare un museo privilegiando aspetti diversi: ordinando i risultati solo per pertinenza oppure anche in base alla distanza da una posizione o in base ai feedback degli altri utenti. Queste diverse possibilità si traducono in differenti *query* da utilizzare per interrogare il database e oltretutto una (e solo una) di queste strategie deve essere utilizzata per l'ordinamento dei risultati di ricerca. Il *behavioral pattern* **Strategy**, permette al nostro programma di cambiare in ogni momento (a discrezione dell'utente), l'algoritmo che regola la ricerca dei musei. Si concretizza nell'interfaccia **SearchStrategy** che viene implementata da **ScoreStrategy**, **LocationStrategy** e **RatingStrategy**.

- **Builder**
  Nel programma che abbiamo realizzato, la funzione principale è la visualizzazione delle informazioni di un museo. Queste sono di diversi tipi, e di conseguenza la classe museo presenta un gran numero di attributi (che potrebbero anche aumentare nelle release successive, con l'aumento delle informazioni di interesse). Per creare un oggetto di tipo museo sarebbe stato necessario un *costruttore* che avrebbe preso molti parametri; inoltre necessitavamo anche di un ulteriore costruttore (che avrebbe preso come parametri nome e id del museo per la visualizzazione dei risultati di una ricerca). Abbiamo quindi deciso di adoperare il pattern **Builder** per la realizzazione di questi oggetti complessi. Il Builder che abbiamo utilizzato non è quello classico del **GoF**, bensì si tratta di una *nested class* privata e statica all'interno di Museo, che contiene una serie di metodi *setter*, che, ritornando sempre l'istanza del Builder, permettono tramite **method chaining** di rendere il codice di creazione degli oggetti molto più leggibile.
  Questo tipo di builder viene utilizzato da *Joshua Bloch* nel suo libro *Effective Java*, in particolare nell'*item 2*. Oltre alla leggibilità e la maggiore chiarezza del codice, con il builder evitiamo di creare i cosiddetti **telescopic constructors** e separiamo la logica di creazione degli oggetti dall'eventuale *business logic*.

- **Factory**
  Per la creazione dei vari Gateway abbiamo utilizzato il pattern **abstract factory**. Questo ci permette di raggruppare la *logica di creazione* dei gateway in un solo punto del codice, rendendolo più facile da mantenere. Inoltre abbiamo scelto questo pattern perché permette di raggruppare gli oggetti creati in *famiglie*, nel nostro caso ci interessava dividere i gateway in base al linguaggio supportato dal database che viene utilizzato per memorizzare i dati. Questa suddivisione rende più facili le modifiche, visto che risulta semplice cambiare la factory concreta utilizzata, essendo questa istanziata una sola volta.
  La scelta di usare questo pattern è in parte dovuta alla volontà di mantenere l'applicazione *aperta all'estensione*: noi infatti abbiamo implementato un solo tipo di factory perché al momento abbiamo utilizzato un database relazionale che sfrutta postgreSQL.

- **MVC**
  Per il nostro programma abbiamo realizzato un'interfaccia grafica con java *Swing* e *Swingx*. Data questa scelta abbiamo deciso di adottare il pattern MVC (**model-view-controller**), un *pattern architetturale* per la progettazione e strutturazione di applicazioni di tipo interattivo. L'applicazione risulta divisa in tre parti (corrispondenti nel codice agli omonimi package): il **model** che rappresenta il modello dei dati di interesse per l'applicazione, il **controller** che definisce la logica di controllo e le funzionalità applicative, la **view** che gestisce la logica di presentazione dei dati.
  Il nostro MVC opera in maniera molto semplice: la view rappresenta le pagine che vengono mostrate all'utente e con cui questo può interagire; ad ogni azione dell'utente viene invocato un metodo del controller, che va a sua volta chiama un metodo nel model (o più spesso di un gateway). Quando le operazioni (di modifica o richiesta dati) effettuate sul model sono concluse, il controller ritorna un valore, o semplicemente il controllo, alla view che lo aveva invocato.
  Il *sequence diagram* in figura 10 descrive questo procedimento.

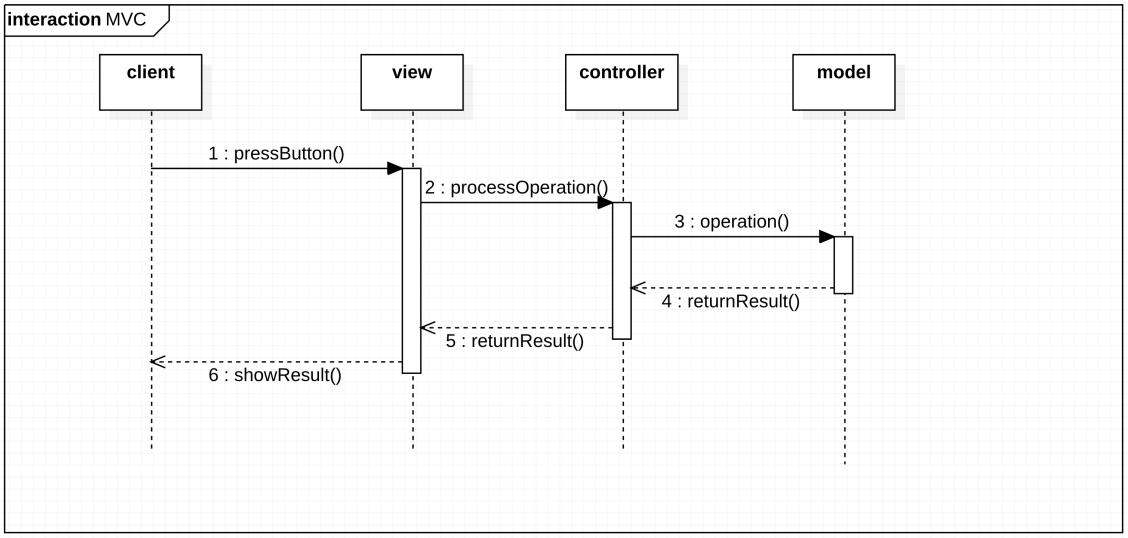

Figura 10: MVC - Sequence Diagram

5 Test

Per lo **unit testing** abbiamo usato il framework **junit 5.0**. Abbiamo creato diverse classi di test per le classi principali che contengono la business logic del programma, in particolare i gateway. Per i test abbiamo usato diverse annotazioni di junit: **@beforeall** e **@afterall** per inserire e eliminare dal database un museo o un utente "test" su cui lavorare, @test per marcare i metodi di test e in alcune classi **@TestMethodOrder(MethodOrderer.OrderAnnotation.class)** e **order()** per indicare l'ordine in cui eseguire i test. Infatti operando con un database l'ordine dei test a volte risulta fondamentale, visto che il test precedente potrebbe lasciare il database in uno stato imprevisto; in ogni caso nel metodo con l'annotazione @afterall ci assicuriamo di eliminare, assieme ai dati "test", ogni modifica fatta dai test stessi.
Abbiamo sfruttato vari **assert** offerti da Junit come **assertEquals(), assertTrue(), assertFalse(), assertThrows()**. I test che abbiamo effettuato sono di vari tipi: funzionali, di performance, di sicurezza.

Le classi di test sono:

- ConnectionPoolTest
- UserGatewayTest
- MuseumGatewayTest
- EventGatewayTest
- ReviewGatewayTest
- ReportGatewayTest
- MuseumListProxy
- SearchStrategyProxy

Dopo ogni classe mostriamo a titolo d'esempio uno dei test presenti al suo interno.

5.1 ConnectionPoolTest

I test relativi alla classe ConnectionPool, verificano la corretta creazione e gestione delle connessioni. In primo luogo viene verificato che effettivamente la creazione renda disponibili il numero prestabilito di connessioni (in questo caso 5) ed anche che una seconda erronea chiamata al metodo *create* non abbia effetti indesiderati (nella pratica viene ignorata). Abbiamo poi verificato che non vengano rese disponibili contemporaneamente più connessioni di quelle create e che il rilascio sia corretto. Infine viene testato il metodo di chiusura del ConnectionPool, che procede al *rilascio* e alla *chiusura* di tutte le connessioni.

```
@Test
public void connectionTest() throws SQLException {
  ArrayList<Connection> connections = new ArrayList<>();
  for (int i = 0; i < 5; i++) {
    connections.add(ConnectionPool.getConnection());
    Assertions.assertEquals(ConnectionPool.getSize(), 5);
  }

  //getting a connection when all the other ones are used
  Assertions.assertThrows(RuntimeException.class, ConnectionPool::
    getConnection);

  for (Connection conn : connections) {
    ConnectionPool.releaseConnection(conn);
    Assertions.assertEquals(ConnectionPool.getSize(), 5);
  }

}
```

5.2 UserGatewayTest

In questa classe di test vengono utilizzati un utente e un museo di prova, gestiti nei metodi marcati con @beforeall e @afterall. Abbiamo testato tre funzionalità importanti, ossia *login*, *registrazione* e *accesso* ad un museo. Oltre ai test di tipo puramente funzionale abbiamo scelto di aggiungere anche dei **security test** per testare l'immunità dei nostri metodi dal **SQL injection**. Questa è una tecnica molto diffusa per attaccare software che sfruttano database relazionali, quindi abbiamo ritenuto opportuno testarla.
Per proteggerci dal problema, abbiamo usato i **PreparedStatement** di JDBC che a differenza dei semplici **Statement** eseguono il *parsing* delle variabili passate, evitanto così esiti inaspettati.

```
  @Test
  public void accessTest() throws SQLException, SizeLimitExceededException {
    UserGateway userGateway = gatewayFactory.getUserGateway();
    User user = userGateway.login(email, password);
    int userId = user.getId();
    userGateway.access(museumId, userId, new Date());
    Assertions.assertEquals(1, userGateway.getAllTimeAccesses(museumId));
    Assertions.assertEquals(1, userGateway.getDailyAccesses(museumId));
    Assertions.assertEquals(1, userGateway.getAllTimeAccesses(museumId));
    Date date = new GregorianCalendar(2016, Calendar.MARCH, 11).getTime();
    userGateway.access(museumId, userId, date);
    Assertions.assertEquals(2, userGateway.getAllTimeAccesses(museumId));
    Assertions.assertEquals(1, userGateway.getWeeklyAccesses(museumId));
    Assertions.assertEquals(1, userGateway.getDailyAccesses(museumId));
```

```
    long DAY_IN_MS = 1000 * 60 * 60 * 24;
    date = new Date(System.currentTimeMillis() - (3 * DAY_IN_MS));
    userGateway.access(museumId, userId, date);
    Assertions.assertEquals(3, userGateway.getAllTimeAccesses(museumId));
    Assertions.assertEquals(2, userGateway.getWeeklyAccesses(museumId));
    Assertions.assertEquals(1, userGateway.getDailyAccesses(museumId));
  }
```

5.3 ReportGatewayTest

Nella classe testiamo, oltre alla già citata SQL injection, il corretto funzionamento dei metodi per immagazzinare e estrarre i report dal database. In particolare abbiamo verificato che il meccanismo di report agisca in accordo con i nostri requisiti funzionali: i report non approvati vengono visualizzati solo dai proprietari, quelli approvati solo dagli admin ed entrambi gli utenti possono eliminare i report.

```
1  @Test
2  public void reportsAdminTest() throws SQLException {
3    ReportGateway reportGateway = gatewayFactory.getReportGateway();
4    int nRep = reportGateway.getAllReports().size();
5
6    //testing that admin doesn't see not approved reports
7    reportGateway.report(1, museumId, testString, false);
8    Assertions.assertEquals(reportGateway.getAllReports().size(), nRep);
9
10   //testing that admin sees approved reports
11   reportGateway.report(1, museumId, testString, true);
12   Assertions.assertEquals(reportGateway.getAllReports().size(), nRep + 1);
13   int id1 = reportGateway.getAllReports().get(reportGateway.getAllReports().
     size() - 1).getId();
14   Assertions.assertEquals(testString, reportGateway.getAllReports().get(
     reportGateway.getAllReports().size() - 1).getDescription());
15
16   //testing that admin correctly removes reports
17   reportGateway.report(1, museumId, testString, true);
18   Assertions.assertEquals(reportGateway.getAllReports().size(), nRep + 2);
19   int id2 = reportGateway.getAllReports().get(reportGateway.getAllReports().
     size() - 1).getId();
20   reportGateway.resolveReport(id1);
21   Assertions.assertEquals(reportGateway.getAllReports().size(), nRep + 1);
22   reportGateway.resolveReport(id2);
23   Assertions.assertEquals(reportGateway.getAllReports().size(), nRep);
24 }
```

5.4 MuseumGatewayTest

Con i test di questa classe verifichiamo principalmente casi limite o errori che possono avvenire durante la ricerca di un museo. Ad esempio testiamo che una query malformata non presenti alcun risultato o che una query generica presenti almeno un risultato. Inoltre testiamo anche il meccanismo delle prenotazioni, per controllare che vengano effettivamente immagazzinate e visualizzate. Anche qui viene verificata immunità a SQL injection.

```
1  @Test
2  public void searchMuseumsTest() throws SQLException {
3    //Test void results
```

```
4       Assertions.assertThrows(NullPointerException.class, () -> gatewayFactory.
        getMuseumGateway().searchMuseums("skdcsjndscdkslcdjchsd", null));
5       //Test normal research
6       JSONArray a1 = gatewayFactory.getMuseumGateway().searchMuseums("museo",
        null);
7       if (a1 != null) {
8         Assertions.assertFalse(a1.isEmpty());
9       }
10    }
11
12    @Test
13    public void getMuseumTest() {
14      //Test functional requirement of getMuseums()
15      Museum museum = null;
16      int id = testInt;
17      try {
18        museum = gatewayFactory.getMuseumGateway().getMuseum(id, new ArrayList
        <>(), new ArrayList<>(), 0);
19      } catch (SQLException e) {
20        System.out.println("This museum is not in the database, change id");
21      }
22      if (museum != null) {
23        Assertions.assertEquals(id, museum.getMuseumId());
24      }
25      //Test that if id provided is not in db the method throws an exception
26      Assertions.assertThrows(IllegalArgumentException.class, () ->
        gatewayFactory.getMuseumGateway().getMuseum(-1, new ArrayList<>(), new
        ArrayList<>(), 0));
27    }
```

5.5 ReviewGatewayTest

Qui verifichiamo che i metodi relativi alle recensioni operino come stabilito: ogni recensione deve essere memorizzata con i giusti parametri e legata al rispettivo museo. Testiamo anche la correttezza delle statistiche sulle recensioni e la SQL injection.

```
1     @Test
2     @Order(1)
3     public void reviewsTest() throws SQLException {
4       ReviewGateway reviewGateway = gatewayFactory.getReviewGateway();
5       reviewGateway.writeReview(museumId, testInt, testString, testInt);
6       ArrayList<Review> reviews = reviewGateway.getReviews(museumId);
7       Assertions.assertEquals(1, reviews.size());
8       Assertions.assertEquals(testString, reviews.get(0).getText());
9       Assertions.assertEquals(testInt, reviews.get(0).getScore());
10      int randomScore = ThreadLocalRandom.current().nextInt(1, 5);
11      reviewGateway.writeReview(museumId, testInt, testString, randomScore);
12      reviews = reviewGateway.getReviews(museumId);
13      Assertions.assertEquals(2, reviews.size());
14      Assertions.assertEquals((float) (testInt + randomScore) / 2, reviewGateway
        .getAverageScore(museumId));
15      Assertions.assertEquals(2, reviewGateway.getReviewsNumber(museumId));
16    }
17
18    @Test
19    @Order(2)
20    public void reviewSQLInjectionTest() throws SQLException {
21      gatewayFactory.getReviewGateway().writeReview(museumId, 1, "prova, 1);
        DROP TABLE reviews --", 5);
```

```
22      Assertions.assertNotEquals(0, gatewayFactory.getReviewGateway().getReviews
        (museumId).size());
23    }
```

5.6 EventGatewayTest

I test di questa classe sono analoghi a quelli della sezione precedente ma con gli eventi.

```
1   @Test
2   public void eventsTest() throws SQLException {
3     //creating tomorrow date
4     Date dt = new Date();
5     Calendar c = Calendar.getInstance();
6     c.setTime(dt);
7     c.add(Calendar.DATE, 1);
8     dt = c.getTime();
9     EventGateway eventGateway = gatewayFactory.getEventGateway();
10    //Typology: 1 discount 2 closed 3 else
11    eventGateway.createEvent(museumId, testString, new Date(), dt, 1, 50);
12    Assertions.assertEquals(5, eventGateway.getTicketPrice(museumId));
13    eventGateway.createEvent(museumId, testString, new Date(), dt, 2, 0);
14    Assertions.assertTrue(eventGateway.isClosed(museumId, new Date()));
15    eventGateway.createEvent(museumId, testString, new Date(), dt, 3, 0);
16    ArrayList<Event> events = eventGateway.getEvents(museumId);
17    Assertions.assertEquals(3, events.size());
18    Assertions.assertEquals(testString, events.get(2).getDescription());
19  }
```

5.7 MuseumListProxyTest

Per verificare il corretto funzionamento del MuseumListProxy abbiamo controllato che la richiesta degli stessi identici risultati restituisse effettivamente dati analoghi, mentre utilizzando la stessa query per la ricerca dei musei, ma richiedendo indici differenti (simulando il pulsante nella GUI che permette di ricevere più risultati) i dati presentati differissero in ogni elemento. Prerogativa del nostro Proxy è quella di implementare la *lazy initialization*, perciò è presente un test che verifica che i risultati siano presentati solo quando l'utente ne fa esplicita richiesta.

```
1   @Test
2   public void loadMuseumsTest() throws SQLException {
3     MuseumListProxy mlp = new MuseumListProxy();
4     mlp.getMuseums("musei per ciechi", null, false);
5     ArrayList<Museum> m1 = mlp.loadMuseums(0, 10);
6     Assertions.assertEquals(10, m1.size());
7     ArrayList<Museum> m3 = mlp.loadMuseums(0, 10);
8     for (int i = 0; i < 10; i++) {
9       Assertions.assertEquals(m1.get(i).getName(), m3.get(i).getName());
10    }
11    ArrayList<Museum> m2 = mlp.loadMuseums(10, 20);
12    for (int i = 0; i < 10; i++) {
13      Assertions.assertNotEquals(m1.get(i).getName(), m2.get(i).getName());
14    }
15    m1.addAll(m2);
16    Assertions.assertEquals(20, m1.size());
17  }
```

5.8 SearchStrategyTest

I test sulle strategia di ricerca verificano che ognuno dei possibili algoritmi di ricerca operi nel modo corretto: Pertanto ci aspettiamo che la *LocationStrategy* ordini i risultati per distanza (due query identiche ma in cui vengono indicate località differenti come punto di partenza tendenzialmente non restituiscono lo stesso ordine dei risultati) e che se non viene riconosciuta la località, ci sia un *fallback* alla ricerca solo basato sull'affinità della query. Per quanto riguarda la *RatingStrategy* abbiamo controllato che effettivamente l'ordinamento dei risultati fosse basato sul punteggio medio delle recensioni.
L'ultimo test interroga un requisito di **performance** dell'applicazione: ogni query (anche una adeguatamente lunga) deve essere eseguita entro i 5 secondi.

```
1  @Test
2  void performanceTest() {
3    System.out.println("Checking if the strategies meet the performance
     constraint of 5 seconds per query...");
4    gatewayFactory.getMuseumGateway().setStrategy(new ScoreStrategy());
5    Assertions.assertTimeout(Duration.ofSeconds(5), () -> {
6      gatewayFactory.getMuseumGateway().searchMuseums("query molto lunga che
     puo' aumentare il tempo di esecuzione", null);
7    });
8
9    gatewayFactory.getMuseumGateway().setStrategy(new LocationStrategy());
10   Assertions.assertTimeout(Duration.ofSeconds(5), () -> {
11     gatewayFactory.getMuseumGateway().searchMuseums("query molto lunga che
     puo' aumentare il tempo di esecuzione", "firenze");
12   });
13
14   gatewayFactory.getMuseumGateway().setStrategy(new RatingStrategy());
15   Assertions.assertTimeout(Duration.ofSeconds(5), () -> {
16     gatewayFactory.getMuseumGateway().searchMuseums("query molto lunga che
     puo' aumentare il tempo di esecuzione", null);
17   });
18 }
```