

PoolManager

Alessandro Aldo Raoul Bonciani

June 2024



Contents

1	Introduzione	4
1.1	Obiettivo e descrizione del progetto	4
1.2	Struttura e pratiche utilizzate	4
2	Progettazione	5
2.1	Use case diagram	5
2.2	Use case template	6
2.3	Mockups	10
2.4	Class diagram	13
2.5	ER Diagram e modello relazionale	15
2.6	Navigation Diagram	17
3	Implementazione	19
3.1	Domain Model	19
3.1.1	Object	19
3.1.2	Chair	19
3.1.3	Deckchair	19
3.1.4	Sunbed	19
3.1.5	Table	19
3.1.6	Umbrella	19
3.1.7	TimeRecord	19
3.1.8	Postation	19
3.1.9	User	20
3.1.10	Location	20
3.1.11	Reservation	20
3.2	ORM	20
3.2.1	UserDAO	20
3.2.2	AdminDAO	20
3.2.3	ObjectDAO	21
3.2.4	TimeRecordDAO	21
3.2.5	LocationDAO	21
3.2.6	PostationDAO	22
3.2.7	ReservationDAO	22
3.2.8	AdminController	22
3.2.9	UserController	23
3.2.10	ReserveController	23
3.2.11	ResourcesController	24
3.3	Database	25
3.4	Interfaccia e CLI	26
4	Testing	28
4.1	AdminControllerTest	28
4.2	UserControllerTest	28
4.3	ReserveControllerTest	29
4.4	ResourcesControllerTest	29

4.5	LoginControllerTest	30
-----	-------------------------------	----

1 Introduzione

Elaborato per il superamento dell'esame di Ingegneria del Software, appartenente al modulo Basi di Dati/Ingegneria del Software del corso di laurea Triennale in Ingegneria Informatica presso l'Università degli Studi di Firenze.

Il progetto è stato sviluppato da Alessandro Aldo Raoul Bonciani, matricola 7079209, nel periodo tra maggio - giugno 2024 (a.a. 2023-2024).

Il codice sorgente è disponibile su *GitHub* al seguente indirizzo:
<https://github.com/alebonch/PoolManager>

1.1 Obiettivo e descrizione del progetto

Con questo progetto si sviluppa un programma adibito alla gestione delle prenotazioni di postazioni per la balneazione in un impianto natatorio.

Si ha quindi un sistema in cui, utenti e admin, possono inserire prenotazioni andando a richiedere un numero di risorse per la propria postazione. Inoltre, il sistema di amministrazione, a cui si accede come admin, permette di modificare dati di utenti e la disponibilità delle risorse. Per gestire i dati e salvarli, il programma è stato connesso ad un database.

1.2 Struttura e pratiche utilizzate

Il software è stato sviluppato in Java, mentre per la gestione e il salvataggio dei dati è stato connesso un database PostgreSQL ed è stata utilizzata la libreria JDBC (Java DataBase Connectivity).

Per mantenere una separazione delle responsabilità, la struttura del progetto è stata divisa in tre parti principali: Business Logic, Domain Model e ORM. Questi tre packages si occupano in modo distinto della logica di business, della rappresentazione dei dati e dell'accesso ai dati (Figura 1):

- **Business Logic:** contiene le classi che implementano la logica di business del sistema.
- **Domain Model:** contiene le classi che rappresentano le entità del sistema.
- **ORM:** contiene le classi che implementano l'Object-Relational Mapping. In questo modo è possibile rendere i dati persistenti e recuperarli dal database.

Per utilizzare il software è stata creata un'interfaccia da riga di comando (CLI) che permette di interagire con il sistema in modo semplice e intuitivo.

Gli Use Case Diagram e i Class Diagram seguono lo standard UML (Unified Modeling Language) e sono stati realizzati con il software StarUML.

Le piattaforme e i software utilizzati sono stati:

- **VisualStudio Code:** IDE per lo sviluppo in Java
- **StarUML:** software per la creazione di diagrammi UML

- **PgAdmin**: software per la gestione di database PostgreSQL
- **GitHub**: piattaforma per la condivisione del codice sorgente
- **Lunacy**: software per la creazione di mock-up
- **Draw.io**: sito che è stato utilizzato per creare il navigation diagram.
- **Overleaf.com**: sito per la creazione di file latex

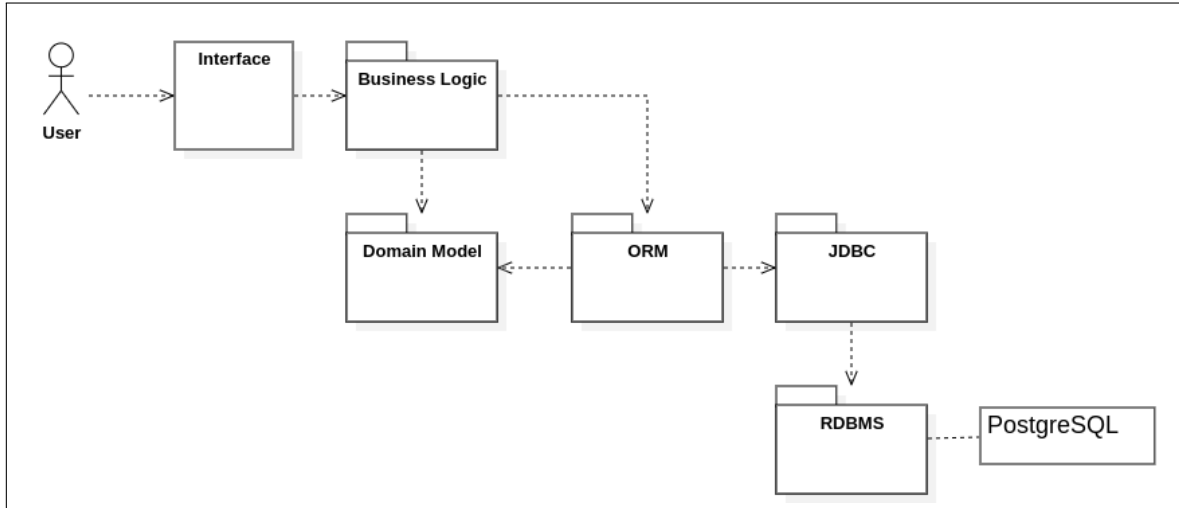


Figure 1: Package Dependency Diagram

2 Progettazione

2.1 Use case diagram

Come precedentemente descritto, il sistema permette agli utenti di effettuare prenotazioni di postazioni in determinati turni di una giornata e di gestire il proprio profilo. La seguente immagine mostra il diagramma dei casi di uso del sistema sia per gli utenti che la parte amministrativa.

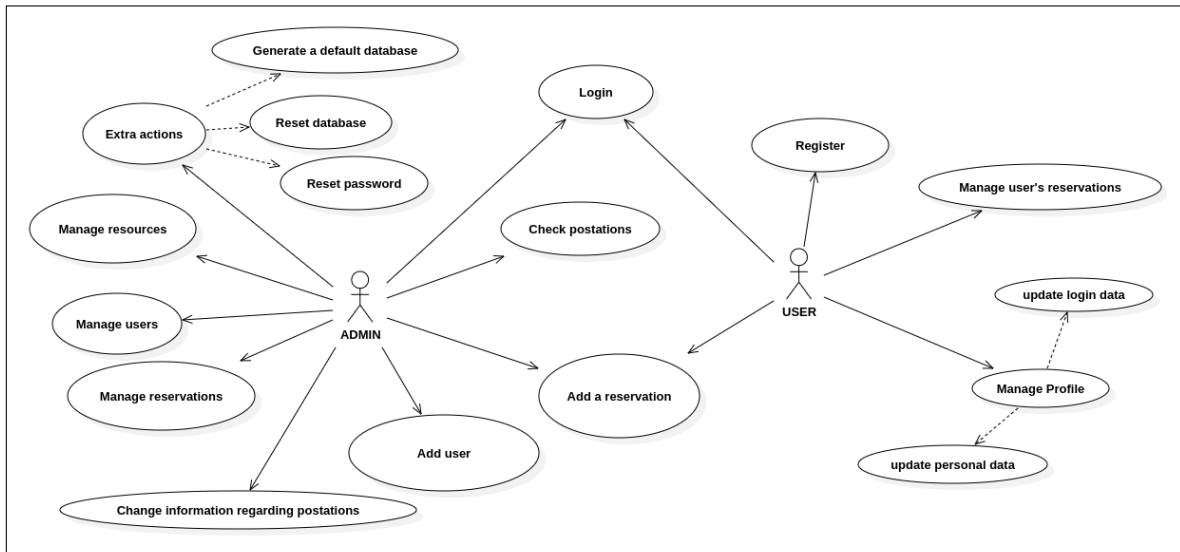


Figure 2: Use Case Diagram

2.2 Use case template

Di seguito sono riportati i template di alcuni dei casi d'uso implementati, in ognuno possiamo trovare: una breve descrizione, il livello del caso d'uso, gli attori coinvolti, le pre-condizioni, le post-condizioni, il flusso base e i flussi alternativi.

Use Case #1	Accesso al sistema (Log-in)
Brief Description	L'utente accede al sistema tramite le proprie credenziali (Mockup Login, Login-ControllerTest)
Level	Function
Actors	Utente, Admin
Pre-conditions	L'utente deve essere nella pagina iniziale di accesso.
Basic Flow	<ol style="list-style-type: none"> 1. L'utente inserisce le proprie credenziali (username e password) 2. L'utente preme il pulsante di accesso 3. Il sistema verifica le credenziali 4. Il sistema autentica l'utente
Alternative Flows	<ol style="list-style-type: none"> 3a. Se le credenziali fornite non sono corrette, il sistema mostra un messaggio di errore all'utente 4a. Se a fare l'accesso è l'admin, il sistema lo reindirizza alla Admin Page (TST #03)
Post-conditions	L'utente è autenticato nel sistema e ha accesso alle funzionalità riservate

Table 1: Use Case Template 1 (Sign in)

Use Case #2	Registrazione nel sistema (Register)
Brief Description	L'utente si registra nel sistema creando un nuovo account (LoginControllerTest)
Level	User Goal
Actors	Utente
Pre-conditions	L'utente deve essere nella pagina iniziale di registrazione
Basic Flow	<ol style="list-style-type: none"> 1. L'utente fornisce i dettagli richiesti per la registrazione 2. L'utente conferma la registrazione 3. Il sistema verifica i dati forniti 4. Il sistema crea un nuovo account per l'utente
Alternative Flows	3a. Se l'utente fornisce dati non validi o se l'email o lo username sono già utilizzati da un altro account, il sistema mostra un messaggio di errore
Post-conditions	L'utente è registrato nel sistema e può accedere utilizzando le credenziali create durante la registrazione

Table 2: Use Case Template 2 (Sign up)

Use Case #3	Creazione di una prenotazione
Brief Description	L'utente prenota una postazione richiedendo un certo numero di risorse. (Mockup Reserve, AdminControllerTest)
Level	User Goal
Actors	Utente, Admin
Pre-conditions	L'utente deve essere nella pagina di gestione delle prenotazioni
Basic Flow	<ol style="list-style-type: none"> 1. L'utente fornisce i dettagli richiesti per la prenotazione 2. Il sistema verifica che la richiesta sia accettabile 3. Il sistema calcola il costo della prenotazione 4. Il sistema crea una nuova prenotazione, una postazione e la inserisce nel database
Alternative Flows	3a. Se l'utente richiede troppe risorse, il sistema genera dei messaggi di errore
Post-conditions	La prenotazione viene inserita nel sistema e il numero di risorse disponibili in quel time record diminuirà in base alla richiesta

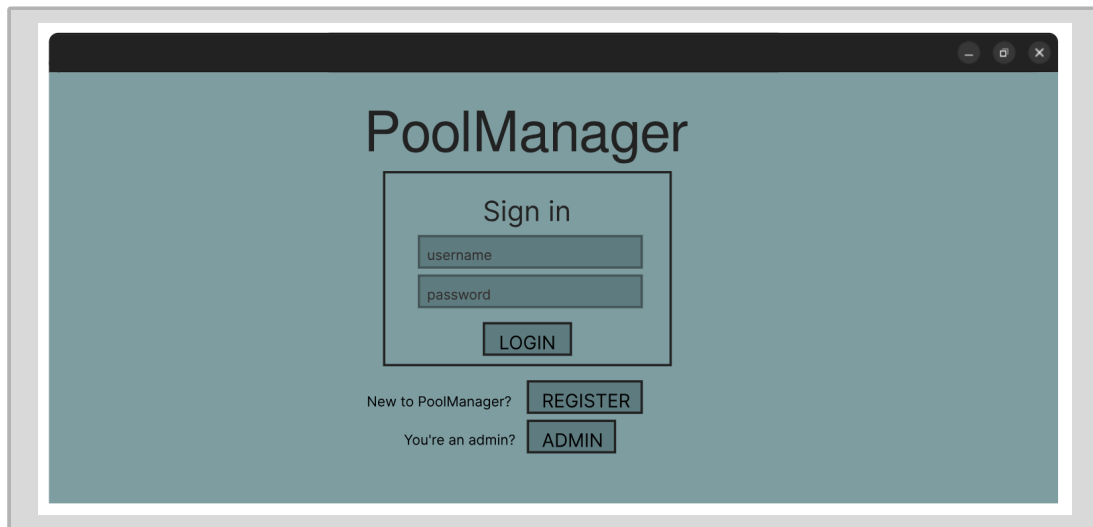
Table 3: Use Case Template 3 (Reserve)

Use Case #4	Ripristino del database
Brief Description	L'Admin esegue l'operazione di ripristino del database (AdminControllerTest)
Level	Function
Actors	Admin
Pre-conditions	L'Admin ha effettuato l'accesso al sistema
Basic Flow	<ol style="list-style-type: none"> 1. L'amministratore seleziona l'opzione per il ripristino del database 2. L'amministratore conferma l'avvio del processo di ripristino 3. Il sistema esegue il ripristino del database 4. Il sistema conferma all'amministratore che il ripristino è stato completato con successo
Alternative Flows	<ol style="list-style-type: none"> 3a. Se il processo di ripristino del database fallisce per qualsiasi motivo, il sistema mostra un messaggio di errore all'amministratore
Post-conditions	Il database è stato ripristinato con successo

Table 4: Use Case Template 4 (Reset database)

2.3 Mockups

Di seguito si inseriscono alcuni possibili Mock-ups relativi alle interfacce grafiche del sistema.



PoolManager

Sign in

username

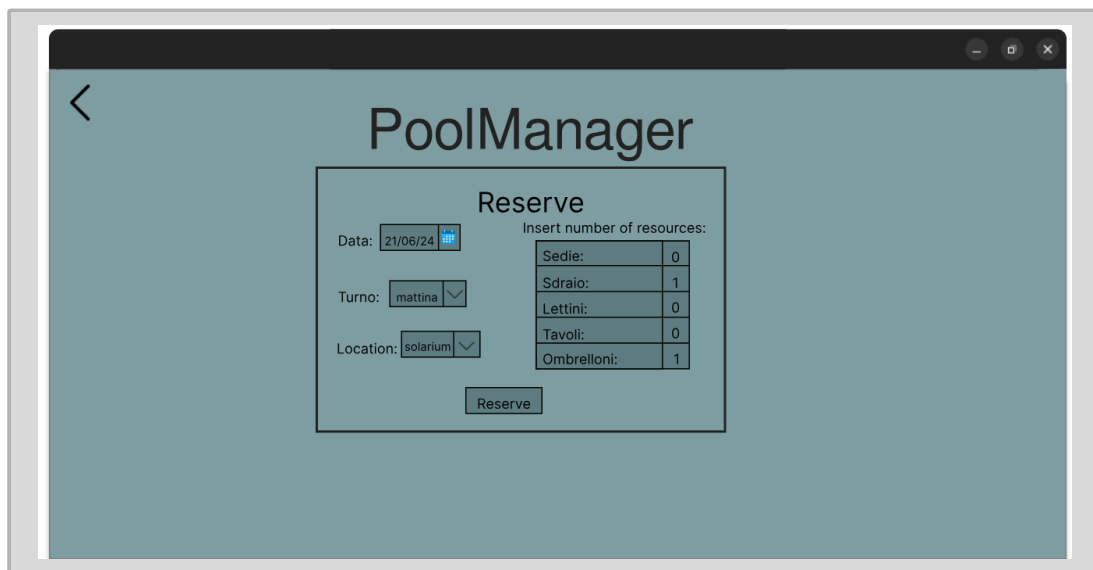
password

LOGIN

New to PoolManager? REGISTER

You're an admin? ADMIN

Figure 3: Prototipo della pagina di login / registrazione



<

PoolManager

Reserve

Data: 21/06/24

Turno: mattina

Location: solarium

Insert number of resources:

Sedie:	0
Sdraio:	1
Lettini:	0
Tavoli:	0
Ombrelloni:	1

Reserve

Figure 4: Prototipo della pagina di prenotazione



Figure 5: Prototipo della pagina di visione delle prenotazioni

2.4 Class diagram

Vista la divisione strutturale in 3 packages, sono stati realizzati 3 diagrammi delle classi distinti, uno per ogni package:

- **Business Logic** (Figura 6): contiene le classi che implementano la logica di business del sistema, ovvero i seguenti controller: quello che gestisce l'accesso e la registrazione dei nuovi utenti (**LoginController**), quello che permette agli utenti aggiornare i propri dati e controllare le proprie prenotazioni (**UserController**), quello che permette di creare e modificare le proprie prenotazioni (**ReserveController**), quello che permette di gestire le risorse disponibili e controllarne la disponibilità (**ResourcesController**), quello che gestisce le azioni dell'admin (**AdminController**)
- **Domain Model** (Figura 7): contiene le classi che rappresentano le entità del sistema, ovvero: **User**, **Object**, **Chair**, **Deckchair**, **Sunbed**, **Table**, **Umbrella**, **Reservation**, **Postation**, **Location** e **TimeRecord**, .
- **ORM** (Figura 8): contiene le classi che implementano l'Object-Relational Mapping, quindi contiene una classe per ogni entità del sistema: **UserDAO**, **ObjectDAO**, **PostationDAO**, **ReservationDAO**, **TimeRecordDAO** e **LocationDAO**. In più contiene anche la classe **AdminDAO** che si occupa della gestione dei dati dell'admin e la classe **ConnectionManager** che si occupa di gestire la connessione al database.

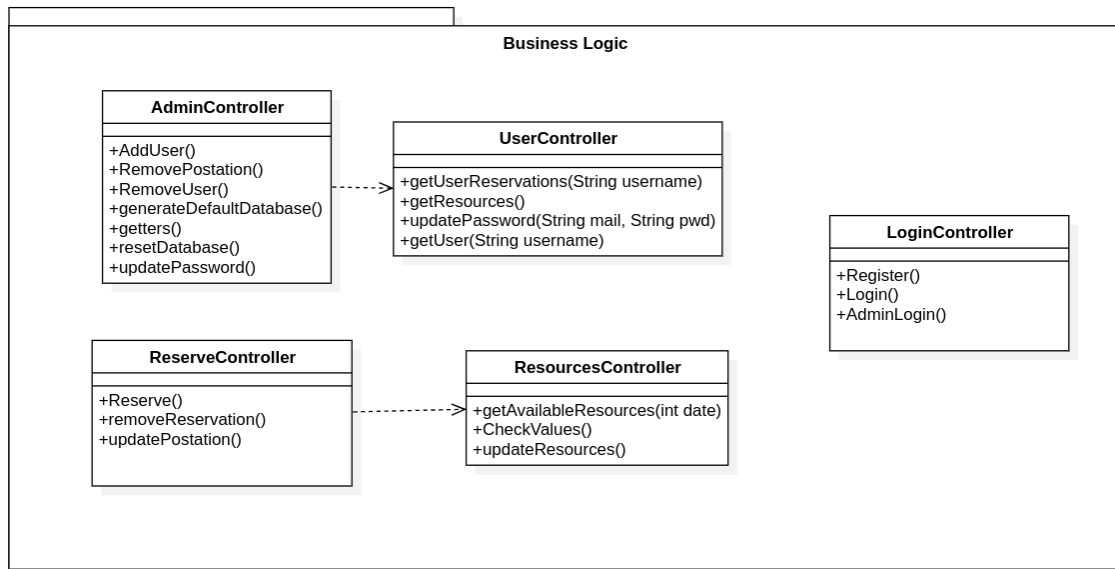


Figure 6: Class Diagram - Business Logic

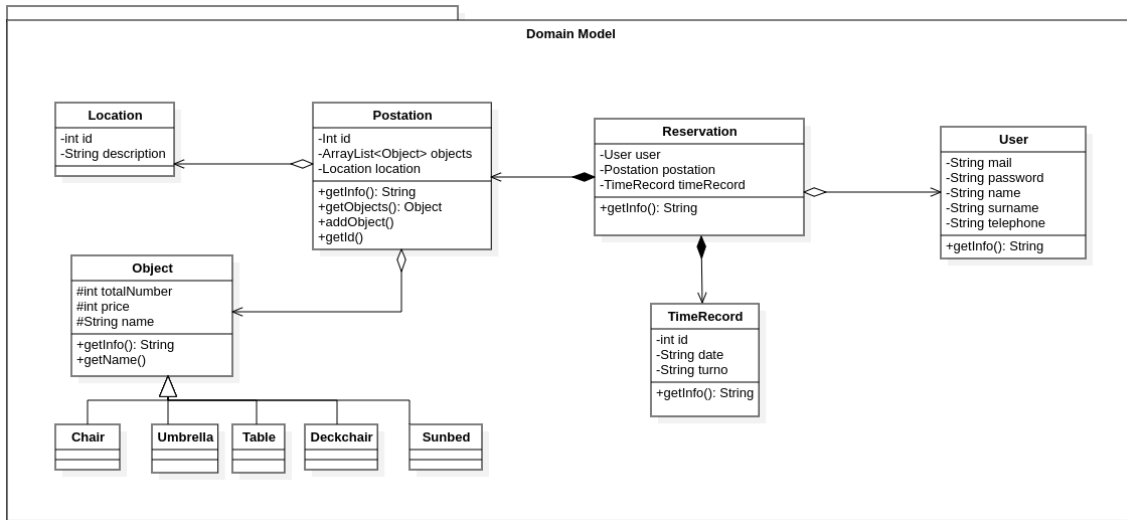


Figure 7: Class Diagram - Domain Model

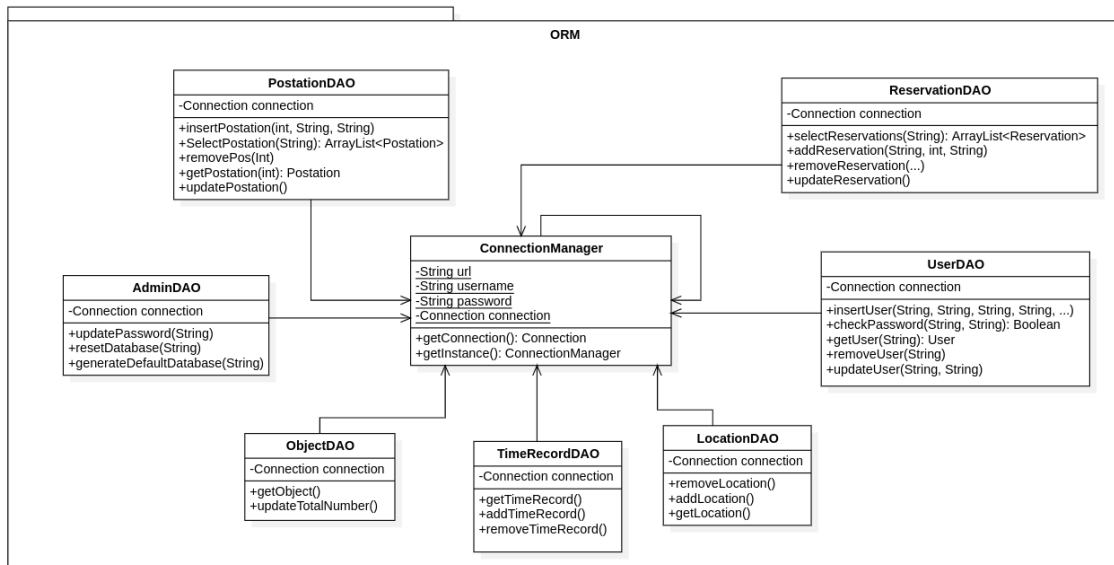


Figure 8: Class Diagram - ORM

2.5 ER Diagram e modello relazionale

Per quanto riguarda la progettazione del database (9), si hanno queste entità:

- **User**: rappresenta l'entità **User**.
- **Postation**: rappresenta l'entità **Postation**. L'ArrayList di oggetti che è presente all'interno della classe (7) è stato inserito inserendo come colonne ogni singolo possibile valore presente tra gli oggetti.
- **Reservation**: rappresenta l'entità **Reservation** e risolve la relazione tra **User**, **TimeRecord** e **Postation**.
- **TimeRecord**: rappresenta l'entità **TimeRecord**.
- **Location**: rappresenta l'entità **Location**
- **Object**: rappresenta l'entità **Object**. A discapito di valori nulli, sono stati inseriti come colonne tutti gli attributi possibili che ogni risorsa può avere.

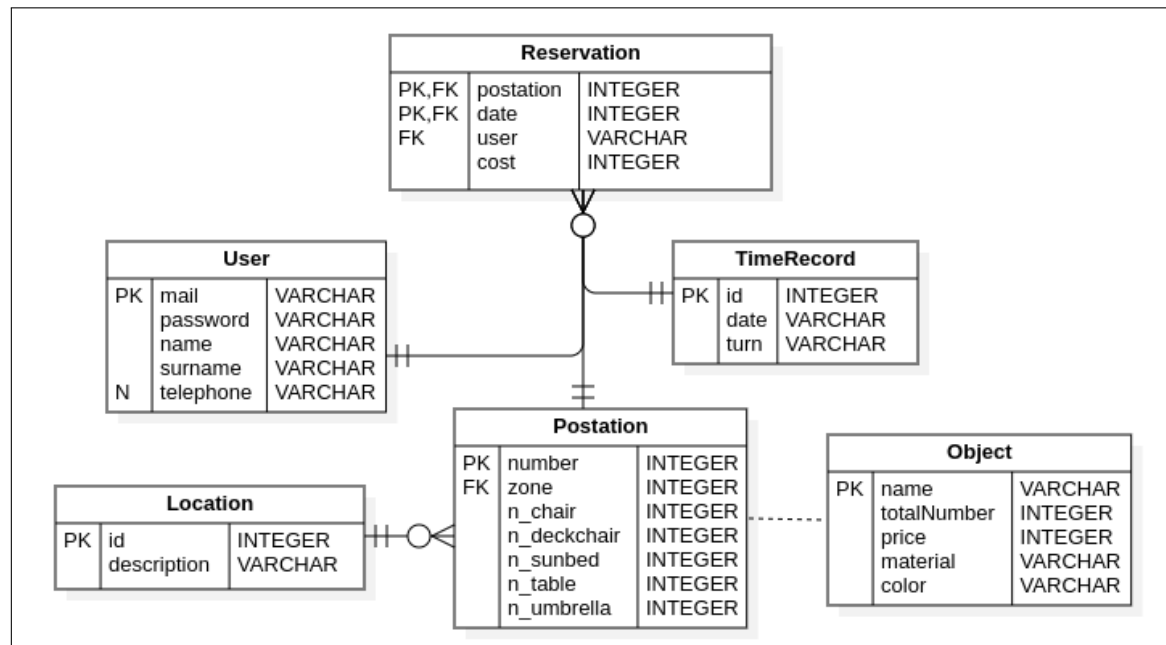


Figure 9: ER Diagram RAW view

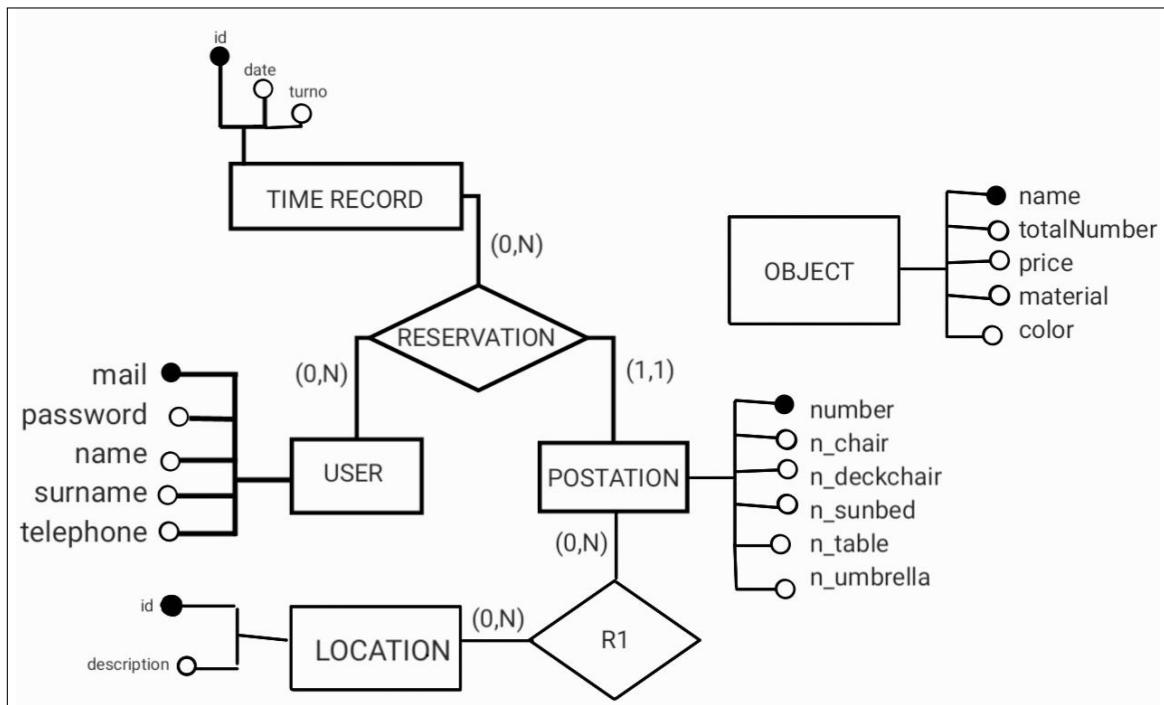


Figure 10: ER Diagram

2.6 Navigation Diagram

Il seguente diagramma (Figura 11) rappresenta quelle che sono le pagine principali del sistema e le possibili azioni che l'utente può compiere. Sono anche rappresentati i modi con cui si può navigare tra le varie pagine.

Alcune delle pagine sono:

- la pagina iniziale **PoolManager**: dove l'utente può registrarsi o effettuare il login.
- **User/Admin Reservation Actions**: in cui si può visualizzare prenotazioni, aggiungerne, rimuoverne o modificarne.
- **Event Editor**: la quale permette di creare un nuovo evento o di modificarne uno esistente.
- **User Profile Actions**: dove è possibile modificare i dati di accesso, cambiare dati personali o controllare i dati attuali.
- **Extra**: dalla quale l'admin può cambiare password o fare il reset del database.

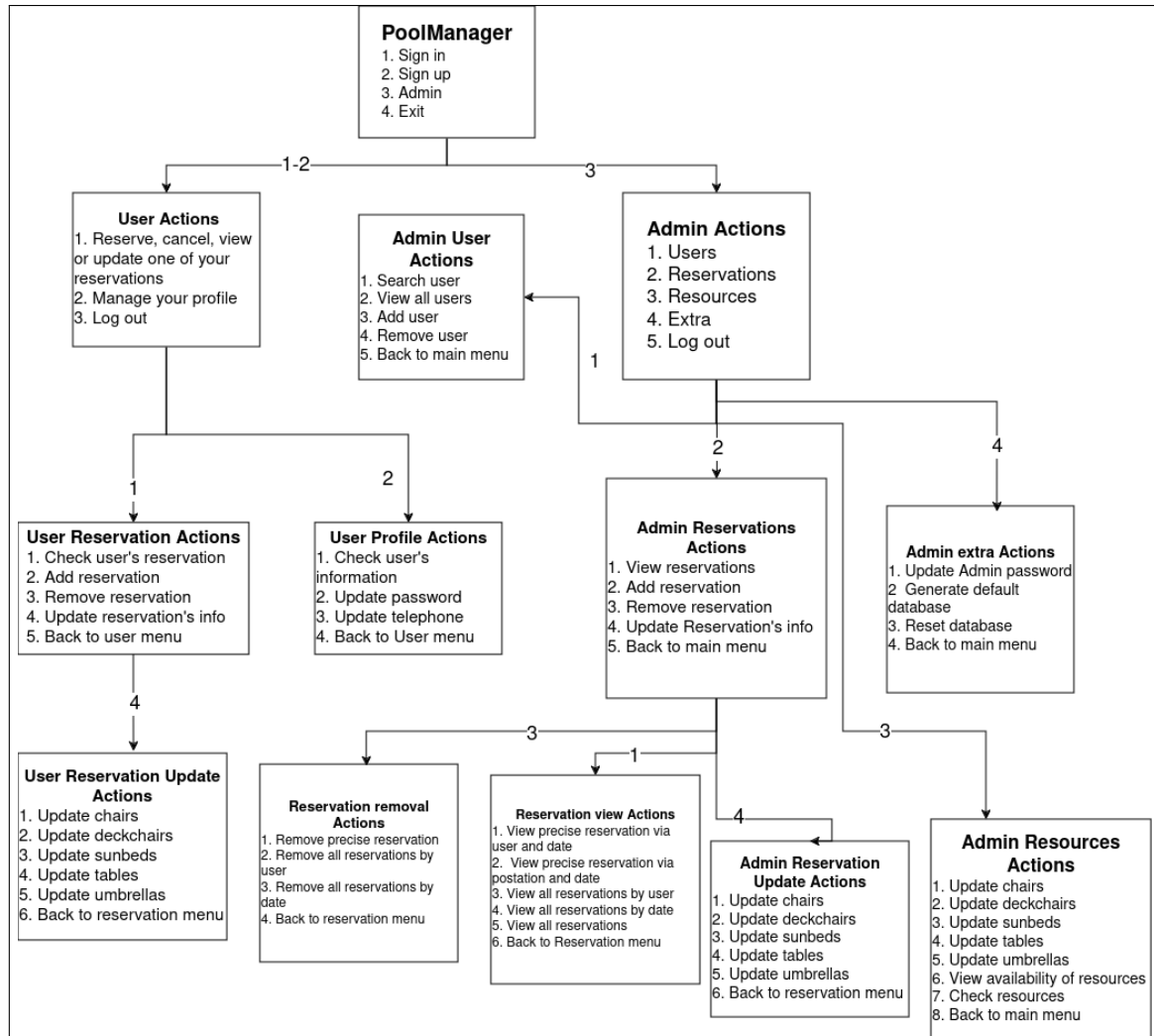


Figure 11: Navigation Diagram

3 Implementazione

L'implementazione dei 3 packages Domain Model, Business Logic e ORM e quella dell'interfaccia sono contenute in `src/main/java`, mentre i file relativi al database sono contenuti in `src/main/sql`.

3.1 Domain Model

Come già accennato nella sezione 2.4, nel package `java.DomainModel` (il percorso è: `src/java/DomainModel`) sono state implementate le classi che rappresentano le entità del sistema.

3.1.1 Object

Classe astratta che rappresenta le risorse, ovvero ogni singolo oggetto possibilmente prenotabile da un utente. I suoi attributi sono: *totalNumber*, *price* e *name*.

3.1.2 Chair

Specifico oggetto derivato dalla classe `Object`. I suoi attributi sono: *color*, *totalNumber*, *price* e *name*.

3.1.3 Deckchair

Specifico oggetto derivato dalla classe `Object`. I suoi attributi sono: *color*, *material*, *totalNumber*, *price* e *name*.

3.1.4 Sunbed

Specifico oggetto derivato dalla classe `Object`. I suoi attributi sono: *material*, *totalNumber*, *price* e *name*.

3.1.5 Table

Specifico oggetto derivato dalla classe `Object`. I suoi attributi sono: *color*, *material*, *totalNumber*, *price* e *name*.

3.1.6 Umbrella

Specifico oggetto derivato dalla classe `Object`. I suoi attributi sono: *color*, *totalNumber*, *price* e *name*.

3.1.7 TimeRecord

Classe che rappresenta il momento per cui viene fissata la prenotazione. I suoi attributi sono: *id*, *date*, *turn*. Dove *turn* è una stringa il cui valore può essere "mattina" o "pomeriggio".

3.1.8 Postation

Classe che rappresenta la postazione richiesta dall'utente nel momento della prenotazione. I suoi attributi sono: *id*, *objects* (un *ArrayList* di oggetti) e una *location*.

3.1.9 User

Un utente già registrato all'interno del sistema. I campi della classe sono: *mail*, *name*, *surname*, *pwd* e *telephone*.

3.1.10 Location

Classe che rappresenta le possibili scelte tra le zone della piscina. I suoi attributi sono: *id*, *description*.

3.1.11 Reservation

Classe che rappresenta la prenotazione. E' formata da: una *postazione*, uno *utente* e un *timerecord*.

3.2 ORM

Come già accennato nella sezione 2.4, nel package `java\ORM` (il percorso è: `src/java/ORM`) sono state implementate le classi che si occupano dell'*Object-Relational Mapping*, ovvero delle operazioni di lettura e scrittura dei dati nel database. Ogni singola classe fuorché **ConnectionManager** permette alle classi del package **java.BusinessLogic** di accedere ai dati salvati nel database e possiede come attributo un'istanza di **ConnectionManager**.

3.2.1 UserDAO

Classe che si occupa della generazione di nuovi utenti attraverso il metodo *insertUser()*, di rimuoverne con *removeUser()*, di aggiornare le informazioni dell'utente e di selezionare utenti. Questa classe si occupa anche del controllo necessario al Login attraverso il metodo *checkPassword()*.

3.2.2 AdminDAO

Classe che si occupa delle Extra actions, ovvero: reset del database, generazione di un database di default e il cambio di password dell'admin.

```

public void resetDatabase(String sql) throws SQLException, ClassNotFoundException {

    PreparedStatement preparedStatement = null;

    try {
        preparedStatement = connection.prepareStatement(sql);
        preparedStatement.executeUpdate();
        System.out.println("\nDatabase reset successfully.");
    } catch (SQLException e) {
        System.err.println("Error: " + e.getMessage());
    } finally {
        if (preparedStatement != null) { preparedStatement.close(); }
    }

}

public void generateDefaultDatabase(String sql) throws SQLException, ClassNotFoundException {

    PreparedStatement preparedStatement = null;

    try {
        preparedStatement = connection.prepareStatement(sql);
        preparedStatement.executeUpdate();
        System.out.println("Default database generated successfully.");
    } catch (SQLException e) {
        System.err.println("Error: " + e.getMessage());
    } finally {
        if (preparedStatement != null) { preparedStatement.close(); }
    }

}

```

Snippet: implementazione delle Extra actions

3.2.3 ObjectDAO

Classe che si occupa della gestione dei dati che riguardano le risorse.

3.2.4 TimeRecordDAO

Classe che si occupa della gestione dei singoli **TimeRecord**, necessaria per calcolare dinamicamente il numero di risorse utilizzate in un dato **TimeRecord**.

3.2.5 LocationDAO

Classe che si occupa dell'inserimento e rimozione delle *Location*.

3.2.6 PostationDAO

Classe che si occupa della gestione delle postazioni, oltre ai metodi per aggiungere, rimuovere, aggiornare e selezionare, è presente un ulteriore metodo *getUsedResources()* necessario al calcolo dinamico delle risorse disponibili.

3.2.7 ReservationDAO

Classe che si occupa della gestione delle prenotazioni, in particolare si è voluto aggiungere diversi metodi di rimozione e selezione di modo da poter andare più o meno nello specifico, in base all'esigenza. E' ovviamente presente anche un metodo per aggiungere una

3.2.8 AdminController

Controller che gestisce le diverse operazioni che possono essere eseguite dall'Actor *Admin*. Si hanno quindi diversi strumenti di selezione per poter controllare tutte le prenotazioni e le postazioni. Inoltre sono qua presenti le Extra Actions. Per alcune operazioni, chiama UserController e ne utilizza i metodi.

```

public void generateDefaultDatabase() throws ClassNotFoundException, SQLException{
    resetDatabase();

    StringBuilder sql_tmp = new StringBuilder();

    try (BufferedReader bufferedReader = new BufferedReader(new FileReader("../sql/default.sql"))) {
        String line;
        while ((line = bufferedReader.readLine()) != null) { sql_tmp.append(line).append("\n"); }
    } catch (IOException e) {
        System.err.println("Error: " + e.getMessage());
        return;
    }

    String sql = sql_tmp.toString();
    AdminDAO adminDAO = new AdminDAO();
    adminDAO.generateDefaultDatabase(sql);
}

public void resetDatabase() throws ClassNotFoundException, SQLException{

    StringBuilder sql_tmp = new StringBuilder();

    try (BufferedReader bufferedReader = new BufferedReader(new FileReader("../sql/reset.sql"))) {
        String line;
        while ((line = bufferedReader.readLine()) != null) { sql_tmp.append(line).append("\n"); }
    } catch (IOException e) {
        System.err.println("Error: " + e.getMessage());
        return;
    }

    String sql = sql_tmp.toString();
    AdminDAO adminDAO = new AdminDAO();
    adminDAO.resetDatabase(sql);

}

```

Figure 13: Snippet: Due delle extra actions

3.2.9 UserController

Controller che gestisce le azioni dell'utente, come il controllo delle proprie prenotazioni e la gestione del proprio profilo.

3.2.10 ReserveController

Controller che si occupa della creazione, rimozione e modifica delle prenotazioni. Chiama **ResourcesController** ogni volta che deve fare un accesso a delle risorse. Inoltre è stato qua inserito il metodo di calcolo del costo di una prenotazione, che viene effettuato alla creazione della prenotazione e ogni volta che essa viene modificata.

```

public void Reserve(int n_chairs,int n_deckchairs,int n_sunbeds, int n_tables, int n_umbrellas, String date,
                    String turn, int location, String username) throws SQLException, ClassNotFoundException {
    ResourcesController resourcesController = new ResourcesController();
    if(resourcesController.CheckValues(n_chairs, n_deckchairs, n_sunbeds, n_tables, n_umbrellas, date)){
        PostationDAO postationDAO = new PostationDAO();
        int id = postationDAO.getCountPostations()+1;
        postationDAO.insertPostation(id, n_chairs, n_tables, n_umbrellas, n_deckchairs, n_sunbeds, location);
        ReservationDAO reservationDAO = new ReservationDAO();
        reservationDAO.addReservation(username, id, TimeRecordFixer(date, turn));
    }
}

```

Figure 14: Snippet: Reserve method

3.2.11 ResourcesController

Controller che si occupa della gestione delle risorse. Il metodo *CheckValues()* viene invocato da **ReserveController** ogni volta che si richiede di modificare la quantità di risorse all'interno di una prenotazione. Sono inoltre presenti i metodi per la modifica delle risorse presenti nell'impianto balneare e un metodo chiamato *getAvailableResources()* che restituisce il numero di risorse ancora disponibili in una certa data.


```

public int[] getAvailableResources(String date) throws SQLException, ClassNotFoundException {
    ObjectDAO objectDAO = new ObjectDAO();
    ArrayList<DomainModel.Object> objects = objectDAO.getAllObjects();
    int[] resources = new int[5];
    ReserveController reserveController = new ReserveController();
    for (DomainModel.Object object : objects) {
        switch (object.getName()) {
            case "chair": resources[0] = object.getNumber(); break;
            case "deckchair": resources[1] = object.getNumber(); break;
            case "sunbed": resources[2] = object.getNumber(); break;
            case "table": resources[3] = object.getNumber(); break;
            case "umbrella": resources[4] = object.getNumber(); break;
        }
    }

    int[] usedResources = new int[5];
    ReservationDAO reservationDAO = new ReservationDAO();
    ArrayList<Reservation> reservations = reservationDAO.SelectReservationsByDate(
        reserveController.TimeRecordFixer(date, "Mattina"));
    reservations.addAll(reservationDAO.SelectReservationsByDate(
        reserveController.TimeRecordFixer(date, "Pomeriggio")));
    PostationDAO postationDAO = new PostationDAO();

    for (Reservation reservation : reservations) {
        int[] currentUsedResources = postationDAO.getUsedResources(reservation.getPosto().getId());
        for (int i = 0; i < 5; i++) {
            usedResources[i] += currentUsedResources[i];
        }
    }
}

```

Figure 15: Snippet: getAvailableResources method

3.3 Database

Come già accennato nella sezione 2.5, il database è stato progettato in modo da rispettare le specifiche del sistema. Sono state create le tabelle `Users`, `Location`, `Object`, `Reservation`, `Postation` e `TimeRecord` e sono state definite come indicato nella Figura 9. I file relativi al database sono contenuti in `src/sql` e i principali sono `reset.sql` e `default.sql`: il primo cancella i vecchi dati con il comando `DROP TABLE IF EXISTS` e ricrea le tabelle secondo la struttura sopra citata, mentre il secondo riempie le tabelle con dei dati di default per testare il funzionamento del sistema.

```

-- Creazione della tabella Utenti
CREATE TABLE Users (
    mail VARCHAR(50) PRIMARY KEY,
    password VARCHAR(50) NOT NULL,
    name VARCHAR(50) NOT NULL,
    surname VARCHAR(50) NOT NULL,
    telephone VARCHAR(50)
);
-- Creazione degli oggetti
CREATE TABLE Object (
    name VARCHAR(50) PRIMARY KEY,
    totalnumber INT NOT NULL,
    price INT NOT NULL,
    material VARCHAR(50),
    color VARCHAR(50)
);
-- Creazione della tabella Locazione
CREATE TABLE Location(
    id INT PRIMARY KEY,
    description VARCHAR(100)
);
-- Creazione della tabella Postazione
CREATE TABLE Postation (
    number INT PRIMARY KEY NOT NULL,
    n_chair INT,
    n_deckchair INT,
    n_sunbed INT,
    n_table INT,
    n_umbrella INT,
    zone INT NOT NULL REFERENCES Location(id)
);

```

Snippet: Database implementation

3.4 Interfaccia e CLI

Come introdotto nella sezione 1.2, è stata realizzata un'interfaccia a riga di comando per il sistema, implementata nel file `Main.java` (il percorso è: `src/java/Main.java`). L'utente può navigare tra le varie pagine e compiere le funzionalità del programma inserendo i vari comandi indicati dal sistema. Prendendo per esempio la pagina iniziale `PoolManager`, l'utente può scegliere di eseguire l'accesso inserendo il comando 1, di registrarsi inserendo il comando 2, di autenticarsi come admin inserendo il comando 3 o di uscire dal programma inserendo il comando 4. Ogni pagina è implementata con un *do-while* e uno *switch-case* per gestire le varie scelte dell'utente, mentre le varie funzionalità sono implementate con metodi specifici delle classi del package `BusinessLogic`. La navigazione tra le pagine avviene tramite chiamate agli stessi metodi della classe `Main`.

```

private static void handleLoginAction() throws ClassNotFoundException, SQLException{
    Scanner scanner = new Scanner(System.in);
    LoginController loginController = new LoginController();
    String input;

    do {

        System.out.println(
            """
            \s
            PoolManager
            1. Sign in
            2. Sign up
            3. Admin
            4. Exit
            \s"""
        );

        input = scanner.nextLine();

        switch (input) {
            case "1" -> {

                Scanner scanner1 = new Scanner(System.in);

                System.out.println("\nMail: ");
                String mail = scanner1.nextLine();
                System.out.println("Password: ");
                String password = scanner1.nextLine();

                User user = loginController.login(mail, password);

                if (user != null)
                    handleUserAction(user);

            }
            case "2" -> {
                System.out.println("\nWelcome!");
                String[] data = register();

                User user = loginController.register(data[0], data[1], data[2], data[3], data[4]);

                if (user != null)
                    handleUserAction(user);

            }
        }
    }
}

```

Snippet: implementazione della pagina iniziale e della funzionalità di login (Sign in)

4 Testing

Data la struttura del codice, è stato deciso di testare i metodi principali delle classi della BusinessLogic, poiché l'implementazione delle classi del DomainModel non ha alcun metodo particolare oltre a *getters* e *setters*. Inoltre i metodi scelti utilizzano metodi di DomainModel e di ORM. Per ogni classe del pacchetto scelto, è stata creata una classe in cui è stato testato almeno un metodo della classe di riferimento. Avendo inoltre implementato tutti i metodi presentati nel Navigation Diagram, è stato deciso di formalizzare un minor numero di test rispetto al numero di metodi effettivamente presenti all'interno delle classi.

4.1 AdminControllerTest

Il test scelto per questo controller riguarda il metodo *generateDefaultDatabase()*, poiché nel farlo è stato possibile testare anche i metodi: *AddUser()*, *getUser()*, *Reserve()* e *getUserReservations()*. Si è quindi controllato che, dopo aver generato il database di default, che si trova al percorso **src/sql/default.sql**, inserendo un nuovo utente non presente inizialmente non ci fossero prenotazioni a suo nome. Infine si è controllato che, inserita una prenotazione, questa risultasse di fatto all'interno del Database.

```
@Test
public void generateDefaultDatabaseTest() throws SQLException, ClassNotFoundException{
    AdminController adminController = new AdminController();
    String username = "tommasofredducci@example.com";
    adminController.generateDefaultDatabase();
    adminController.AddUser(username, "Tommaso", "Fredducci", "123", "redelghiaccio");
    assertTrue(adminController.getUser(username)!=null);
    ArrayList<Reservation> reservations = adminController.getUserReservations(username);
    assertEquals(0, reservations.size());
    ReserveController reserveController = new ReserveController();
    reserveController.Reserve(1, 1, 1, 1, 1, "2024-06-19", "Mattina", 1, username);
    assertEquals(1, adminController.getUserReservations(username).size());
    adminController.generateDefaultDatabase();
}
```

Snippet: test di generateDefaultDatabase()

4.2 UserControllerTest

Si è scelto di testare uno dei metodi di aggiornamento del profilo di uno **User**, prendendo come riferimento il metodo *updatePassword()*, a questa maniera è stato possibile riconfermare il test di *getUser()* e di uno dei *getters* di **User**.

```

@Test
public void updatePasswordTest() throws SQLException, ClassNotFoundException{
    String username = "user1@example.com";
    String password = "password1";
    AdminController adminController = new AdminController();
    adminController.generateDefaultDatabase();
    User user = adminController.getUser(username);
    assertEquals(password, user.getPwd());
    UserController userController = new UserController();
    password = "iceking";
    userController.updatePassword(username, password);
    user = adminController.getUser(username);
    assertEquals(password, user.getPwd());
    adminController.generateDefaultDatabase();
}

```

Snippet: test di updatePassword()

4.3 ReserveControllerTest

Avendo già implicitamente testato il metodo *Reserve()* all'interno del test di **AdminControllerTest**, è stato qua invece implementato il test di *removeReservation()*. Nello specifico, partendo dalla generazione di un database di default, si salva il numero di prenotazioni totali all'interno di un intero *size*, si procede a rimuovere una prenotazione che conosciamo essere presente all'interno di **src/sql/default.sql** e si confronta il numero di prenotazioni dopo la rimozione con quello precedente.

```

@Test
public void removeReservationTest() throws SQLException, ClassNotFoundException{
    AdminController adminController = new AdminController();
    adminController.generateDefaultDatabase();
    int size= adminController.getAllReservations().size();
    ReserveController reserveController = new ReserveController();
    reserveController.removeReservation("user2@example.com", 2);
    ArrayList<Reservation> reservations = adminController.getAllReservations();
    assertEquals(size-1, reservations.size());
    adminController.generateDefaultDatabase();
}

```

Snippet: test di removeReservation()

4.4 ResourcesControllerTest

Per questo controller è stato testato il metodo di controllo dei vincoli sulle risorse *CheckValues()*, andando ad inserire valori che entrano all'interno dei vincoli e valori che fuoriescono dai vincoli.

```

@Test
public void CheckValuesTest() throws SQLException, ClassNotFoundException{
    ResourcesController resourcesController = new ResourcesController();
    assertTrue(resourcesController.CheckValues(0, 1, 2, 3, 4, "2024-06-19"));
    assertFalse(resourcesController.CheckValues(5000, 200, 100, 100, 0, "2024-06-19"));
}

```

Snippet: Test di checkValues()

4.5 LoginControllerTest

Per quanto riguarda questo controller, sono stati invece testati i metodi *Login()* e *Register()* andando a controllare che entrambi i metodi risolvessero i propri flussi nella maniera corretta. Nel primo caso, si controlla sia quando si ha un accesso con credenziali corrette, sia quando si cerca di effettuare l'accesso con delle credenziali che non lo sono. Nel caso di *Register()*, invece, si è semplicemente controllato che il metodo restituisse correttamente lo **User** inserito.

```

@Test
public void LoginTest() throws SQLException, ClassNotFoundException{
    // Test case 1: this is a known user in the database
    String username = "user1@example.com";
    String password = "password1";

    LoginController loginController = new LoginController();

    try {
        User user = loginController.login(username, password);
        assertNotNull(user);
    } catch (SQLException | ClassNotFoundException e) {
        System.err.println(e.getMessage());
    }

    // Test case 2: Incorrect password
    String username2 = "user2@example.com";
    String password2 = "Password2";

    try {
        User user2 = loginController.login(username2, password2);
        assertNull(user2);
    } catch (SQLException | ClassNotFoundException e) {
        System.err.println(e.getMessage());
    }
}

```

Snippet: test di login()

```

@Test
public void RegisterTest() throws SQLException, ClassNotFoundException{
    String username = "tommasofredducci@example.com";
    String password = "redelghiaccio";
    String name = "Tommaso";
    String surname = "Fredducci";
    AdminController adminController = new AdminController();
    adminController.generateDefaultDatabase();
    LoginController loginController = new LoginController();

    try {
        User user = loginController.register(username, password, name, surname);
        assertNotNull(user);
    } catch (SQLException | ClassNotFoundException e) {
        System.err.println(e.getMessage());
    }
    adminController.removeUser(username);
}

```

Snippet: test di register()