



UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Relazione di Ingegneria del Software

Autori:

Alberto Pizzi
Tommaso Ciccotti

N° Matricola:

(A.P.) 7073782

(T.C.) 7073874

Corso principale:
Ingegneria del Software

Docente corso:
Enrico Vicario

Indice

1 Introduzione	5
1.1 Statement	5
1.1.1 Obiettivo	5
1.1.2 Descrizione	5
1.2 Struttura del progetto	5
1.3 Tecnologie utilizzate	6
2 Progettazione	7
2.1 Tabella dei permessi	7
2.2 Use Case Diagram	8
2.3 Use Case Templates	9
2.4 Mockups	17
2.4.1 User Side	17
2.4.2 Manager Side	23
2.4.3 Owner Side	25
2.5 Navigation Diagrams	32
2.5.1 User-Manager's Diagram	32
2.5.2 Owner's Diagram	33
2.6 Class Diagrams	34
2.6.1 DomainModel	34
2.6.2 BusinessLogic	35
2.6.3 ORM	35
2.6.4 GUI	36
2.7 Database	36
2.7.1 Modello ER	37
2.7.2 Modello Relazionale	38
3 Implementazione	39
3.1 Domain Model	39
3.1.1 GroupMember	39
3.1.2 Group	39
3.1.3 NotificationType	40
3.1.4 Notification	40
3.1.5 Observer	40
3.1.6 Observable	40
3.1.7 Reservation	41
3.1.8 Person	41
3.1.9 User	41
3.1.10 Owner	42
3.1.11 WorkingHours	42
3.1.12 Sport	42
3.1.13 Invite	42
3.1.14 Product	42
3.1.15 Creator	42
3.1.16 InviteSender	42
3.1.17 NotificationSender	43
3.1.18 Field	43
3.1.19 Facility	43
3.2 Business Logic	44
3.2.1 PasswordEncoder	44
3.2.2 ProfileController	44
3.2.3 UserProfileController	45
3.2.4 OwnerProfileController	45
3.2.5 PersonController	45
3.2.6 UserActionsController	48
3.2.7 ManagerOwnerManagementController	51
3.2.8 OwnerManagementController	51
3.2.9 NotificationController	51
3.2.10 AccessStrategy	53

3.2.11	UserAccess	53
3.2.12	OwnerAccess	54
3.2.13	AccessController	54
3.2.14	SessionController	54
3.3	Object-Relational Mapping (ORM)	55
3.3.1	ConnectionManager	55
3.3.2	ConnectionHolder	56
3.3.3	FacilityDAO	56
3.3.4	ManagesDAO	56
3.3.5	NotificationDAO	56
3.3.6	PersonDAO	57
3.3.7	UserDAO	57
3.3.8	OwnerDAO	57
3.3.9	WorkingHoursDAO	57
3.3.10	SportDAO	57
3.3.11	InviteDAO	58
3.3.12	IsPartDAO	58
3.3.13	FieldDAO	58
3.3.14	GroupDAO	58
3.3.15	ReservationDAO	58
3.4	GUI	59
3.4.1	FieldFormManagementController	59
3.4.2	BookFieldController	59
3.4.3	BookFieldManagerController	59
3.4.4	BookFieldOwnerController	59
3.4.5	FieldDetail	59
3.4.6	FieldDetailManagerController	59
3.4.7	FieldDetailUserController	59
3.4.8	FieldDetailOwnerController	60
3.4.9	GroupItemController	60
3.4.10	YourGroupsController	60
3.4.11	InviteItemController	60
3.4.12	YourInvitesController	60
3.4.13	NotificationItem	60
3.4.14	NotificationItemController	60
3.4.15	NotificationItemOwnerController	60
3.4.16	Notifications	60
3.4.17	NotificationsController	61
3.4.18	NotificationsOwnerController	61
3.4.19	Menu	61
3.4.20	MenuController	61
3.4.21	MenuOwnerController	61
3.4.22	ModifyReservationController	61
3.4.23	ModifyReservationManagerController	61
3.4.24	ModifyReservationOwnerController	61
3.4.25	SelectGuestsPaneController	61
3.4.26	ManageGuestsManagerPaneController	62
3.4.27	ManageGuestsUserPaneController	62
3.4.28	ManagementButtonsController	62
3.4.29	ReservationItemController	62
3.4.30	ReservationItemManagerController	62
3.4.31	ReservationItemOwnerController	62
3.4.32	ReservationsController	62
3.4.33	SceneController	62
3.4.34	AccessGui	63
3.4.35	Login	63
3.4.36	LoginOwner	63
3.4.37	LoginUser	63
3.4.38	SignUp	63
3.4.39	SignUpOwner	63
3.4.40	SignUpUser	63

3.4.41	Announcement	63
3.4.42	AnnouncementOwner	63
3.4.43	AnnouncementManager	64
3.4.44	MediaManagerController	64
3.4.45	Reservations	64
3.4.46	ReservationsManagerController	64
3.4.47	ReservationsOwnerController	64
3.4.48	ReservationItems	64
3.4.49	ReservationItemsManagerOwner	64
3.4.50	ReservationItemOwnerController	64
3.4.51	FacilityForm	64
3.4.52	FieldForm	65
3.4.53	HomeOwnerController	65
3.4.54	HomeUserController	65
3.4.55	UpdateProfileController	65
3.4.56	UpdateAddress	65
3.4.57	UpdateAddressOwnerController	65
3.4.58	UpdateAddressUserController	65
3.4.59	UpdateEmail	65
3.4.60	UpdateEmailOwnerController	65
3.4.61	UpdateEmailUserController	65
3.4.62	UpdatePassword	66
3.4.63	UpdatePasswordOwnerController	66
3.4.64	UpdatePasswordUserController	66
3.4.65	UpdateUsername	66
3.4.66	UpdateUsernameOwnerController	66
3.4.67	UpdateUsernameUserController	66
3.4.68	AddManagersController	66
3.4.69	ModifyWorkingHoursController	66
3.4.70	ModifyFieldController	66
3.4.71	ModifyFacilityController	66
3.4.72	NewSportController	67
3.4.73	NewFieldController	67
3.4.74	NewFacilityController	67
3.4.75	NewWorkingHoursController	67
3.4.76	FieldChoice	67
3.4.77	FieldChoiceOwnerController	67
3.4.78	FieldChoiceManagerController	67
3.4.79	FieldItem	67
3.4.80	FieldItemUserController	67
3.4.81	FieldChoiceItem	67
3.4.82	FieldChoiceItemOwnerController	68
3.4.83	FieldChoiceItemManagerController	68
3.4.84	FacilityDetail	68
3.4.85	FacilityDetailManagerController	68
3.4.86	FacilityDetailOwnerController	68
3.4.87	FacilityChoice	68
3.4.88	FacilitiesController	68
3.4.89	FacilityChoiceOwnerController	68
3.4.90	FacilityChoiceManagerController	68
3.4.91	FacilityItem	68
3.4.92	FacilityItemController	69
3.4.93	FacilityChoiceItem	69
3.4.94	FacilityChoiceItemOwnerController	69
3.4.95	FacilityChoiceManagerController	69
3.5	Database	69
3.5.1	Modelli	69
3.5.2	Triggers	69
3.5.3	Transactions	71

4 Testing	73
4.1 DomainModelTest	73
4.1.1 GeneralTest	73
4.1.2 GroupTest	73
4.1.3 ReservationTest	73
4.1.4 FacilityTest	73
4.1.5 InviteSenderTest	73
4.2 BusinessLogicTest	73
4.2.1 GeneralBSTest	74
4.2.2 NotificationControllerTest	75
4.2.3 PersonControllerTest	75
4.2.4 UserActionControllerTest	75
4.2.5 ProfileController	75
4.2.6 UserProfileController	75
4.2.7 OwnerProfileController	76
4.2.8 OwnerManagementControllerTest	76
4.2.9 ManagerOwnerManagementControllerTest	76
4.2.10 AccessOwnerTest	76
4.2.11 AccessUserTest	76
4.3 ORMTest	76
4.3.1 GeneralDAOTest	77
4.3.2 FacilityDAOTest	77
4.3.3 ManagesDAOTest	77
4.3.4 PersonDAOTest	77
4.3.5 OwnerDAOTest	77
4.3.6 UserDAOTest	77
4.3.7 WorkingHoursDAOTest	78
4.3.8 FieldDAOTest	78
4.3.9 ReservationDAOTest	78
4.3.10 GroupDAOTest	78
4.3.11 InviteDAOTest	78
4.3.12 IsPartDAOTest	78
4.3.13 NotificationDAOTest	78
4.3.14 SportDAOTest	78
5 Bibliografia	79
5.1 Utilizzo di AI	79

1 Introduzione

Elaborato di tipologia 2 per il superamento dell'esame di Ingegneria del Software (modulo Basi di Dati / Ingegneria del Software) e Laboratorio di Informatica del corso di Laurea Triennale in Ingegneria Informatica dell'Università degli Studi di Firenze.

Il software è stato disegnato, progettato e sviluppato da Alberto Pizzi (m. 7073782) e Tommaso Ciccotti (m. 7073874) tra settembre 2024 e aprile 2025 (a.a. 2024/2025).

La repository **GitHub** del codice sorgente si trova all'indirizzo: https://github.com/alberto-pizzi/elaborato_swe

1.1 Statement

Statement consegnato a maggio 2024.

1.1.1 Obiettivo

Realizzazione di un **applicativo software** per la **gestione** di impianti sportivi poli-funzionali orientato ai gestori/proprietari di essi ed ai clienti. La sua funzione principale è quella di permettere ai gestori/proprietari di gestire le prenotazioni ed i campi, e ai clienti di prenotare i campi. L'applicativo, inoltre, presenta anche una funzione di **matching** che permette di organizzare anche partite fra persone estranee tra loro.

1.1.2 Descrizione

Esistono 3 tipologie di utenti, ciascuna con permessi e interfacce differenti:

- Cliente (**User**)
- Gestore (**Manager**)
- Proprietario (**Owner**)

Il proprietario può avere più impianti, così come un manager. Il proprietario può aggiungere, modificare o cancellare i propri impianti. Le varie tipologie di utenti si devono registrare (o accedere) per poter utilizzare l'applicativo. Ogni proprietario di un impianto può indicare dei manager per la gestione dei singoli impianti, oppure rimuoverli. Il proprietario può anche aggiungere o rimuovere dei campi di un determinato impianto di sua proprietà. I clienti possono effettuare prenotazioni di campi e possono modificarle o cancellarle solo se effettuate da loro stessi oppure se sono capo gruppo. Inoltre, è presente anche uno storico delle richieste inviate. I manager e i proprietari possono modificare o cancellare tutte le prenotazioni che riguardano impianti che li competono. Se un cliente vuole organizzare una partita e non raggiunge il numero necessario di partecipanti ad un determinato sport, il sistema permette di ricercare altre persone (tra quelle iscritte) tramite un sistema di notifica e di invio di inviti, se deciso in fase di prenotazione. Il proprietario può gestire il costo orario per i vari campi, così che il sistema possa informare i clienti del costo, calcolandolo.

1.2 Struttura del progetto

La struttura del progetto è divisa nei seguenti packages in modo tale da mantenere la separazione delle responsabilità e dei permessi di accesso:

- **GUIControl**: contiene tutti i singoli GUI Controller richiesti da ogni schermata (file) FXML. Infatti ogni file FXML ha associato *un ed un solo* GUI Controller.
- **BusinessLogic**: contiene le classi che implementano la logica di business del sistema.
- **DomainModel**: contiene le classi che rappresentano le entità del sistema.
- **ORM**: contiene le classi che permettono di interfacciarsi con le relative tabelle del database, ovvero di DAOs.

Il software si interfaccia con l'utilizzatore tramite un'interfaccia grafica creata grazie a **JavaFX**, un framework per la creazione di interfacce grafiche (GUI) basato sul linguaggio Java.

Nella Figura 1 è possibile vedere come i vari packages interagiscono tra loro, incluso l'attore.

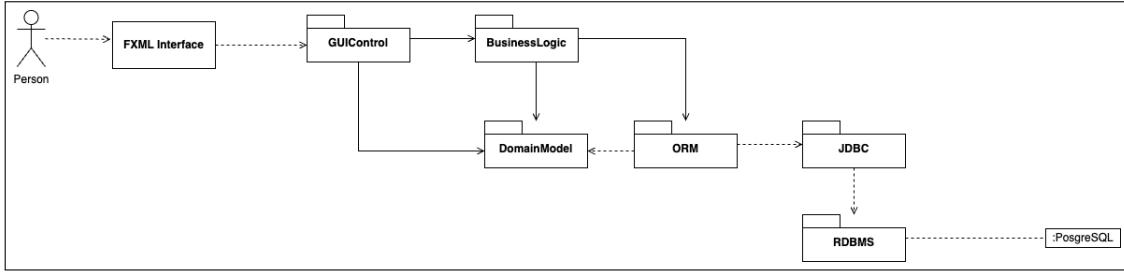


Figura 1: Packages Dependency Diagram

1.3 Tecnologie utilizzate

Tecnologie di sviluppo:

- **Java 19**: linguaggio di programmazione utilizzato per sviluppare il software.
- **JavaFX**: framework utilizzato per creare le interfacce grafiche (GUI).
- **JUnit 5**: libreria di test utilizzata per testare il codice.
- **Mockito 5**: libreria utilizzata da alcune classi di test per controllare le dipendenze durante il testing, utile per assegnare comportamenti specifici e gestire le azioni sui database tramite i DAOs.
- **Apache Commons Lang 3**: libreria utilizzata assieme a **Serializable** di **java.io** per implementare le deep copy in modo rapido.

Strumenti di sviluppo:

- **JetBrains IntelliJ**: IDE di sviluppo.
- **GitHub**: piattaforma di versionamento del codice usata per gestire il codice sorgente.
- **SceneBuilder**: software utilizzato per progettare le interfacce grafiche basate su JavaFX.
- **Draw.io**: software utilizzato per disegnare i diagrammi ER.
- **StarUML**: software utilizzato per disegnare i diagrammi UML.
- **Figma**: software utilizzato per realizzare i disegni dei mockups.

Gestione dati:

- **PostgreSQL**: sistema di gestione di database relazionale usato per la gestione e il salvataggio dei dati.
- **PgAdmin**: software di gestione del database utilizzato per amministrare PostgreSQL.
- **JDBC** (Java DataBase Connectivity): libreria utilizzata per interfacciarsi al database PostgreSQL da Java.

2 Progettazione

2.1 Tabella dei permessi

Permessi:	User	Manager	Owner
Accept Invite	✓	✓	✗
Add Facility	✗	✗	✓
Add Field	✗	✗	✓
Add Reservation (forced)	✗	✓	✓
Add Reservation (standard)	✓	✓	✗
Attach Manager to Facility	✗	✗	✓
Delete Facility	✗	✗	✓
Delete Field	✗	✗	✓
Delete own profile	✓	✓	✓
Delete Reservation	●	✓	✓
Detach Manager to Facility	✗	✗	✓
Edit Facility	✗	✗	✓
Edit Field	✗	✗	✓
Edit Group participants	●	✓	✓
Edit Reservation (forced)	✗	✓	✓
Edit Reservation (standard)	●	●	✗
Join Group	✓	✓	✗
Leave Group	✓	✓	✗
Search other players while making a Reservation (only Matched)	✓	✓	✓
Send announcement	✗	✓	✓
Send Invites from Reservation	✓	✓	✓
Switch matched/not matched	✗	✗	✗
Update personal profile data	✓	✓	✓
View and Delete own Notifications	✓	✓	✓
View own managed Facilities	✗	✓	✓
View own managed Fields and related Reservations	✗	✓	✓

Figura 2: Tabella dei permessi.

Legenda:

- non permesso
- permesso
- permesso sotto determinate condizioni

2.2 Use Case Diagram

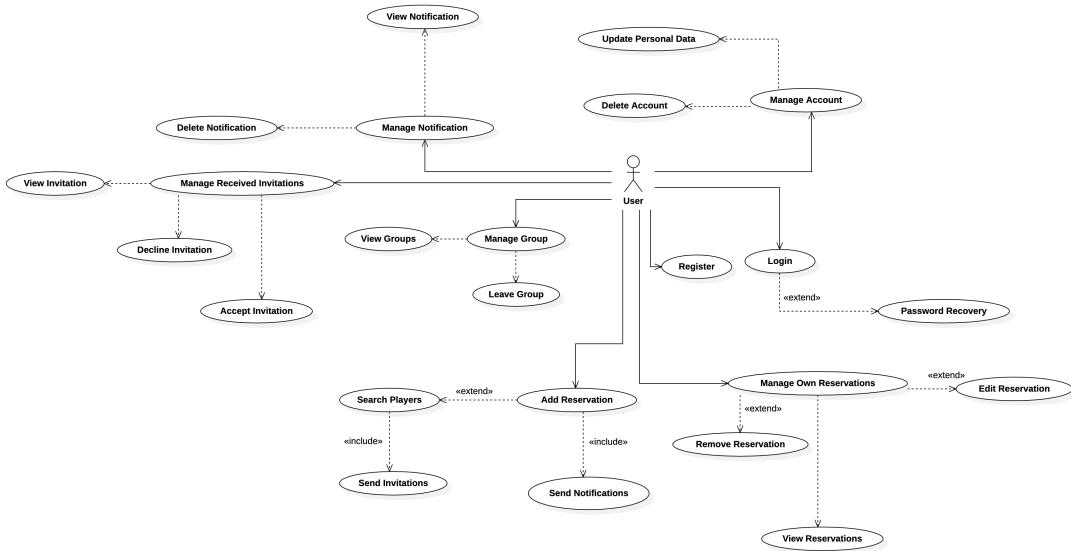


Figura 3: Use Case Diagram di **User**

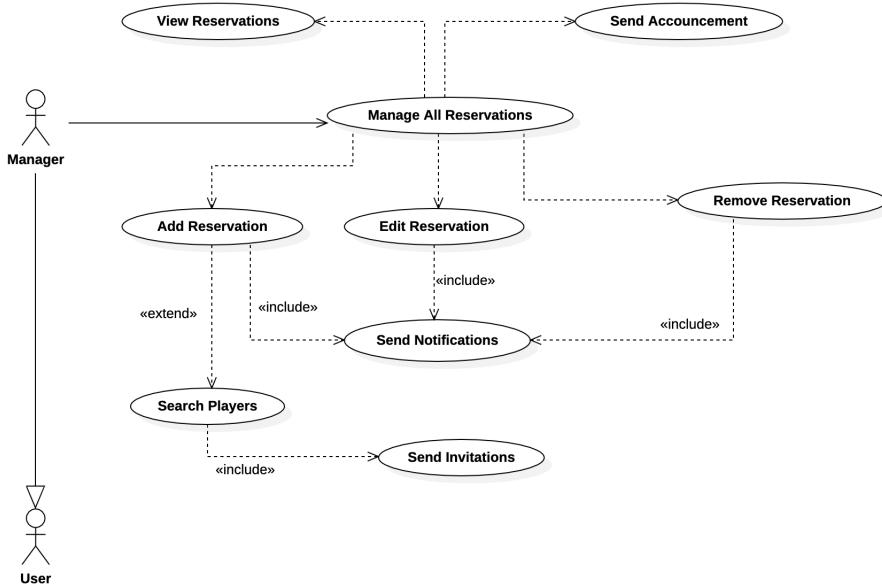


Figura 4: Use Case Diagram di **Manager**

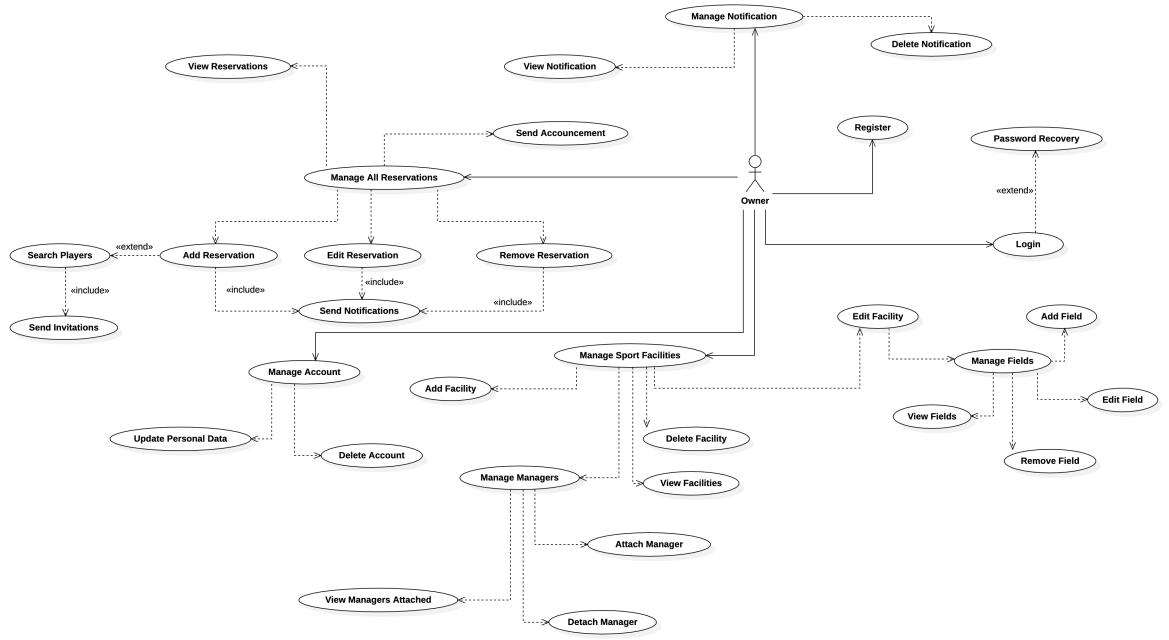


Figura 5: Use Case Diagram di **Owner**

2.3 Use Case Templates

Use Case #1	Registrazione nel sistema (Sign up)
Brief Description	L'utente crea un nuovo account nel sistema (MK Fig. 7 e 26)
Level	User Goal
Actors	Utente, Proprietario
Pre-conditions	L'utente deve essere nella pagina iniziale di registrazione
Basic Flow	<ol style="list-style-type: none"> 1. L'utente compila i campi richiesti per la registrazione 2. L'utente conferma la registrazione 3. Il sistema controlla i dati forniti 4. Il sistema crea un nuovo account per l'utente
Alternative Flows	<p>1a. Il Proprietario ha cliccato sul pulsante dedicato alla registrazione dei proprietari</p> <p>1a.1. Il sistema lo porta al form per la registrazione dedicata ai proprietari.</p> <p>3a. L'utente fornisce dati non validi o se l'email o lo username sono già utilizzati da un altro account</p> <p>3a.1. Il sistema mostra un messaggio di errore e l'utente controlla i dati</p>
Post-conditions	L'utente o il Proprietario è registrato nel sistema e può accedere utilizzando le credenziali inserite in fase di registrazione.

Tabella 1: Use Case Template (Sign up)

I Gestori sono anche Utenti, come indicato nello Use Case Diagram.

Use Case #2	Accesso al sistema (Log in)
Brief Description	L'utente accede al sistema tramite le proprie credenziali (MK Fig. 6 e 27)
Level	User Goal
Actors	Utente, Gestore, Proprietario
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 controlla le credenziali 4. Il sistema autentica l'utente
Alternative Flows	<p>1a. Il Proprietario ha cliccato sul pulsante dedicato al login dei proprietari</p> <p>1a.1. Il sistema porta il proprietario alla pagina dedicata al login dei proprietari</p> <p>1b. La password è stata dimenticata</p> <p>1b.1. Il sistema porta l'utente alla pagina dedicata alla reimpostazione</p> <p>3a. Le credenziali fornite non sono corrette</p> <p>3a.1. Il sistema mostra un messaggio di errore all'utente e l'utente controlla i dati</p>
Post-conditions	L'utente o il Proprietario è autenticato nel sistema e ha accesso alle proprie funzionalità

Tabella 2: Use Case Template (Log in)

Use Case #3	Accettare un invito
Brief Description	L'utente partecipa ad un invito esistente (MK Fig. 15)
Level	User Goal
Actors	Utente, Gestore
Pre-conditions	L'utente deve essere nella pagina degli inviti, e deve fare una richiesta al sistema per partecipare ad un evento
Basic Flow	<ol style="list-style-type: none"> 1. L'utente seleziona un evento disponibile 2. L'utente può selezionare, attraverso un popup, se invitare ulteriori utenti (se consentito) 3. L'utente conferma la partecipazione 4. Il sistema mostra un messaggio di avvenuta partecipazione.
Alternative Flows	<p>4a. Il sistema mostra un messaggio di errore in caso di mancata partecipazione</p> <p>4a.1. Il sistema rimane immutato</p>
Post-conditions	L'invito è accettato e la partecipazione è visibile nella sezione Groups.

Tabella 3: Use Case Template (Accept invitation)

Use Case #4	Prenotazione campo (Cliente)
Brief Description	L'utente crea una nuova prenotazione nel sistema (MK Fig 10)
Level	User goal
Actors	Utente
Pre-conditions	Il sistema deve essere nella pagina relativa al campo o alla panoramica dei campi.
Basic Flow	<ol style="list-style-type: none"> 1. L'utente seleziona l'opzione per creare una nuova prenotazione 2. L'utente inserisce le informazioni richieste (durata, data, ora, numero di persone...) 3. L'utente conferma
Alternative Flows	<p>2a. L'utente inserisce dettagli non validi</p> <p>2a.1. Il sistema mostra un messaggio di errore e chiede di correggere i dettagli</p> <p>3a. L'utente annulla l'operazione.</p> <p>3a.1. L'utente viene portato alla pagina precedente e l'operazione viene annullata</p>
Post-conditions	La prenotazione viene creata con successo ed i relativi inviti sono stati inviati (se necessario). I campi di input vengono puliti.

Tabella 4: Use Case Template (Create a reservation)

Use Case #5	Modifica prenotazione (Cliente)
Brief Description	L'utente modifica una prenotazione effettuata senza ricerca dei giocatori (MK Fig 13)
Level	User goal
Actors	Utente
Pre-conditions	Il sistema deve essere nella pagina relativa alla prenotazione (o alla panoramica) e deve essere entro il limite di preavviso minimo.
Basic Flow	<ol style="list-style-type: none"> 1. L'utente seleziona l'opzione per modificare la prenotazione 2. L'utente modifica i parametri desiderati 3. L'utente conferma le modifiche
Alternative Flows	<p>3a. L'utente annulla le modifiche.</p> <p>3a.1. L'utente viene portato alla pagina precedente e la prenotazione non subisce modifiche.</p>
Post-conditions	La prenotazione viene modificata con successo ed i relativi inviti (e notifiche) sono stati inviati, se necessario.

Tabella 5: Use Case Template (Edit a reservation)

Use Case #6	Elimina prenotazione (Cliente)
Brief Description	L'utente elimina una prenotazione effettuata (senza ricerca di altri giocatori)(MK Fig. 12)
Level	User goal
Actors	Utente
Pre-conditions	Il sistema deve essere nella pagina relativa alla prenotazione (o alla panoramica) e deve essere entro il limite di preavviso minimo.
Basic Flow	<ol style="list-style-type: none"> 1. L'utente seleziona l'opzione per eliminare la prenotazione 2. L'utente conferma l'eliminazione 3. L'utente viene portato alla panoramica
Alternative Flows	<p>2a. L'utente annulla l'operazione.</p> <p>2a.1. L'utente viene portato alla pagina precedente e la prenotazione non subisce modifiche.</p>
Post-conditions	Il sistema rimuove la prenotazione selezionata e notifica i Gestori e il Proprietario della cancellazione

Tabella 6: Use Case Template (Delete a reservation)

Use Case #7	Abbandona gruppo (Cliente)
Brief Description	L'utente abbandona il gruppo (MK Fig. 14)
Level	User goal
Actors	Utente
Pre-conditions	Il sistema deve essere nella pagina relativa alla panoramica dei gruppi.
Basic Flow	<ol style="list-style-type: none"> 1. L'utente seleziona l'opzione per abbandonare il gruppo 2. L'utente conferma l'abbandono
Alternative Flows	<p>2a. L'utente annulla l'operazione.</p> <p>2a.1. L'utente viene portato alla pagina precedente e la prenotazione non subisce modifiche.</p> <p>2b. Il gruppo è vuoto.</p> <p>2b.1. Il gruppo e la relativa prenotazione vengono eliminati.</p> <p>2c. L'utente è capogruppo.</p> <p>2c.1. Viene assegnato un successore come capogruppo.</p>
Post-conditions	Il sistema rimuove l'Utente e gli altri membri del gruppo senza account. Il sistema notifica gli altri Utenti dell'abbandono

Tabella 7: Use Case Template (Leave a group)

Use Case #8	Aggiungi impianto
Brief Description	Il proprietario può aggiungere un impianto sportivo a quelli di sua proprietà (MK 30)
Level	User goal
Actors	Proprietario
Pre-conditions	Il sistema deve essere nella pagina relativa alla panoramica degli impianti sportivi in possesso del Proprietario.
Basic Flow	<ol style="list-style-type: none"> 1. Il proprietario seleziona l'opzione per aggiungere un nuovo impianto 2. Il proprietario inserisce i dati richiesti 3. Il proprietario può nominare dei gestori 4. Il proprietario aggiunge dei campi sportivi 5. Il proprietario conferma le informazioni inserite tramite l'apposito pulsante
Alternative Flows	<p>2a. Il proprietario annulla l'operazione.</p> <p>2a.1. Il proprietario viene portato alla pagina precedente e l'impianto non viene creato</p> <p>5a. I dati inseriti non sono validi</p> <p>5a.1. Il sistema avvisa il proprietario tramite un messaggio di errore.</p>
Post-conditions	Il sistema aggiunge l'impianto.

Tabella 8: Use Case Template (Add Facility)

Use Case #9	Modifica impianto
Brief Description	Il proprietario può modificare un impianto sportivo tra quelli di sua proprietà
Level	User goal
Actors	Proprietario
Pre-conditions	Il sistema deve essere nella pagina relativa alla panoramica degli impianti sportivi in possesso del Proprietario o a quella di dettaglio.
Basic Flow	<ol style="list-style-type: none"> 1. Il proprietario seleziona l'opzione per modificare un impianto esistente 2. Il proprietario modifica i dati necessari 3. Il proprietario può modificare gestori 4. Il proprietario può modificare i campi sportivi 5. Il proprietario conferma le modifiche apportate tramite l'apposito pulsante
Alternative Flows	<p>2a. Il proprietario annulla l'operazione.</p> <p>2a.1. Il proprietario viene portato alla pagina precedente e l'impianto rimane invariato</p> <p>5a. I dati inseriti non sono validi</p> <p>5a.1. Il sistema avvisa il proprietario tramite un messaggio di errore.</p>
Post-conditions	Il sistema modifica l'impianto di riferimento con le informazioni inserite dal proprietario.

Tabella 9: Use Case Template (Edit Facility)

Use Case #10	Eliminazione impianto
Brief Description	Il proprietario può eliminare un impianto sportivo tra quelli di sua proprietà (MK Fig 40)
Level	User goal
Actors	Proprietario
Pre-conditions	Il sistema deve essere nella pagina relativa alla panoramica degli impianti sportivi in possesso del Proprietario.
Basic Flow	<ol style="list-style-type: none"> 1. Il proprietario seleziona l'opzione per eliminare un impianto esistente 2. Il proprietario conferma l'eliminazione
Alternative Flows	<p>2a. Il proprietario annulla l'operazione.</p> <p>2a.1. Il proprietario viene portato alla pagina precedente e l'impianto rimane invariato</p>
Post-conditions	il sistema elimina l'impianto sportivo e i campi, i manager e le ore di apertura collegate.

Tabella 10: Use Case Template (Delete Facility)

Use Case #11	Aggiungi campo sportivo
Brief Description	Il proprietario può aggiungere un campo sportivo ad un impianto di sua proprietà (MK Fig. 31)
Level	User goal
Actors	Proprietario
Pre-conditions	Il sistema deve essere nella pagina di modifica o creazione dell'impianto
Basic Flow	<ol style="list-style-type: none"> 1. Il proprietario seleziona l'opzione per aggiungere un nuovo campo, relativo ad un determinato impianto 2. Il proprietario inserisce i dati richiesti 3. Il proprietario conferma le informazioni inserite tramite l'apposito pulsante
Alternative Flows	<p>2a. Il proprietario annulla l'operazione.</p> <p>2a.1. Il proprietario viene portato alla pagina precedente e il campo non viene creato</p> <p>3a. I dati inseriti non sono validi</p> <p>3a.1. Il sistema avvisa il proprietario tramite un messaggio di errore.</p>
Post-conditions	Il sistema aggiunge il campo sportivo e lo associa il campo all'impianto.

Tabella 11: Use Case Template (Add Field)

Use Case #12	Modifica campo sportivo
Brief Description	Il proprietario può modificare un campo sportivo tra quelli di sua proprietà
Level	User goal
Actors	Proprietario
Pre-conditions	Il sistema deve essere nella pagina relativa alla modifica dell'impianto sportivo
Basic Flow	<ol style="list-style-type: none"> 1. Il proprietario seleziona l'opzione per modificare un campo esistente 2. Il proprietario modifica i dati necessari 3. Il proprietario conferma le modifiche apportate tramite l'apposito pulsante
Alternative Flows	<p>2a. Il proprietario annulla l'operazione.</p> <p>2a.1. Il proprietario viene portato alla pagina precedente e l'impianto rimane invariato</p> <p>3a. I dati inseriti non sono validi</p> <p>3a.1. Il sistema avvisa il proprietario tramite un messaggio di errore.</p>
Post-conditions	Il sistema modifica il campo associato all'impianto selezionato con le informazioni inserite dal proprietario.

Tabella 12: Use Case Template (Edit Field)

Use Case #13	Rimuovi campo sportivo
Brief Description	Il proprietario può eliminare un campo sportivo associato ad un impianto di sua proprietà
Level	User goal
Actors	Proprietario
Pre-conditions	Il sistema deve essere nella pagina dei propri impianti sportivi.
Basic Flow	<ol style="list-style-type: none"> 1. Il proprietario seleziona l'opzione per eliminare un campo esistente 2. Il proprietario conferma l'eliminazione tramite l'apposito pulsante
Alternative Flows	<p>2a. Il proprietario annulla l'operazione.</p> <p>2a.1. Non viene eliminato alcun campo</p>
Post-conditions	Il sistema elimina il campo sportivo e gli elementi collegati come le prenotazioni.

Tabella 13: Use Case Template (Remove Field)

Use Case #14	Aggiunta gestori
Brief Description	Il proprietario può aggiungere dei gestori a un impianto sportivo tra quelli di sua proprietà (MK Fig. 33)
Level	User goal
Actors	Proprietario
Pre-conditions	Il sistema deve essere nella pagina relativa alla modifica o creazione dell'impianto sportivo
Basic Flow	<ol style="list-style-type: none"> 1. Il proprietario seleziona l'opzione per aggiungere dei gestori 2. Il proprietario aggiunge i gestori desiderati 3. Il proprietario conferma le modifiche apportate tramite l'apposito pulsante
Alternative Flows	<p>1a. Il proprietario annulla l'operazione.</p> <p>1a.1. Il proprietario viene portato alla pagina precedente e l'impianto rimane invariato</p>
Post-conditions	Il sistema aggiunge i manager all'impianto specificato.

Tabella 14: Use Case Template (Adding managers)

Use Case #15	Eliminazione gestori
Brief Description	Il proprietario può eliminare i gestori di un impianto tra quelli di sua proprietà
Level	User goal
Actors	Proprietario
Pre-conditions	Il sistema deve essere nella pagina relativa alla modifica dell'impianto sportivo
Basic Flow	<ol style="list-style-type: none"> 1. Il proprietario seleziona l'opzione per cancellare determinati gestori 2. Il proprietario conferma le modifiche apportate tramite l'apposito pulsante
Alternative Flows	<p>2a. Il proprietario annulla l'operazione.</p> <p>2a.1. Il proprietario viene portato alla pagina precedente e l'impianto rimane invariato</p>
Post-conditions	Il sistema rimuove i manager selezionati dall'impianto specificato.

Tabella 15: Use Case Template (Removing managers)

Use Case #16	Fare un annuncio
Brief Description	Il proprietario o i gestori possono fare un annuncio che sarà inviato a tutti i partecipanti di una prenotazione a patto che riguardi un impianto di loro competenza (MK Fig. 39)
Level	User goal
Actors	Proprietario e gestori
Pre-conditions	Il sistema deve essere nella pagina relativa alla panoramica delle prenotazioni
Basic Flow	<ol style="list-style-type: none"> 1. Il proprietario o il gestore seleziona l'opzione per effettuare l'annuncio 2. Il proprietario o il gestore inserisce i dati necessari 3. Il proprietario o il gestore conferma l'annuncio tramite l'apposito pulsante
Alternative Flows	<p>2a. Il proprietario annulla l'operazione.</p> <p>2a.1. Il proprietario viene portato alla pagina precedente e l'impianto rimane invariato</p> <p>3a. I dati inseriti non sono validi</p> <p>3a.1. Il sistema avvisa il proprietario tramite un messaggio di errore.</p>
Post-conditions	Il sistema crea l'annuncio e invia la notifica agli utenti che fanno parte della prenotazione interessata.

Tabella 16: Use Case Template (Make announcement)

Use Case #17	Aggiungi prenotazione (Gestore e Proprietario)
Brief Description	Il Gestore o il Proprietario crea un nuova prenotazione nel sistema (MK Fig. 34)
Level	User goal
Actors	Gestore, Proprietario
Pre-conditions	Il sistema deve essere nella pagina relativa alla panoramica di tutti i campi di propria gestione.
Basic Flow	<ol style="list-style-type: none"> 1. Il Gestore o il Proprietario l'opzione per creare una nuova prenotazione. 2. Il Gestore o il Proprietario inserisce le informazioni richieste. 3. Il Gestore o il Proprietario conferma la prenotazione.
Alternative Flows	<p>2a. Il Gestore o il Proprietario inserisce dettagli non validi</p> <p>2a.1. Il sistema mostra un messaggio di errore e chiede di correggere i dettagli</p> <p>3a. Il Gestore o il Proprietario annulla l'operazione.</p> <p>3a.1. Il Gestore o il Proprietario viene portato alla pagina precedente e l'operazione viene annullata</p>
Post-conditions	La prenotazione viene creata con successo ed i relativi inviti (e notifiche) sono stati inviati (se necessario). I campi di input vengono puliti.

Tabella 17: Use Case Template (Create a reservation (forced))

Use Case #18	Modifica prenotazione (Gestore o Proprietario)
Brief Description	Il Gestore o il Proprietario modifica una prenotazione.
Level	User goal
Actors	Gestore, Proprietario
Pre-conditions	Il sistema deve essere nella pagina relativa alla panoramica delle prenotazioni di un determinato campo. (MK Fig. 25 e 45)
Basic Flow	<ol style="list-style-type: none"> 1. Il Gestore o il Proprietario seleziona l'opzione per modificare la prenotazione. 2. Il Gestore o il Proprietario modifica i parametri desiderati. 3. Il Gestore o il Proprietario conferma le modifiche.
Alternative Flows	<p>3a. Il Gestore o il Proprietario annulla le modifiche.</p> <p>3a.1. Il Gestore o il Proprietario viene portato alla pagina precedente e la prenotazione non subisce modifiche.</p>
Post-conditions	La prenotazione viene modificata con successo ed i relativi inviti (e notifiche) sono stati inviati (se necessario).

Tabella 18: Use Case Template (Edit a reservation (Manager and Owner))

Use Case #19	Elimina prenotazione (Gestore e Proprietario)
Brief Description	Il Gestore o il Proprietario elimina una prenotazione (MK Fig. 25 e 45)
Level	User goal
Actors	Gestore, Proprietario
Pre-conditions	Il sistema deve essere nella pagina relativa alla panoramica delle prenotazioni.
Basic Flow	<ol style="list-style-type: none"> 1. Il Gestore o il Proprietario seleziona l'opzione per eliminare la prenotazione 2. Il Gestore o il Proprietario conferma l'eliminazione 3. Il Gestore o il Proprietario viene portato alla panoramica
Alternative Flows	<p>2a. Il Gestore o il Proprietario annulla l'operazione.</p> <p>2a.1. Il Gestore o il Proprietario viene portato alla pagina precedente e la prenotazione non subisce modifiche.</p>
Post-conditions	Il sistema rimuove la prenotazione selezionata e notifica i Gestori ed il Proprietario della cancellazione.

Tabella 19: Use Case Template (Delete a reservation (Manger and Owner))

Use Case #20	Modifica del profilo
Brief Description	La persona può modificare le informazioni nel proprio profilo (MK Fig. 16)
Level	User goal
Actors	Proprietario, Utente, Gestore
Pre-conditions	Il sistema deve essere nella pagina di gestione del profilo
Basic Flow	<ol style="list-style-type: none"> 1. La persona seleziona il campo che vuole modificare del profilo. 2. La persona inserisce i dati aggiornati. 3. La persona conferma le informazioni inserite tramite l'apposito pulsante.
Alternative Flows	<p>3a. I dati inseriti non sono validi</p> <p>3a.1. Il sistema avvisa la persona tramite un messaggio di errore.</p>
Post-conditions	Il sistema aggiorna le informazioni del profilo.

Tabella 20: Use Case Template (Update profile)

2.4 Mockups

In questo capitolo sono presenti alcuni dei mockup più importanti, facendo riferimento alla tabella dei permessi in Figura 2.

2.4.1 User Side

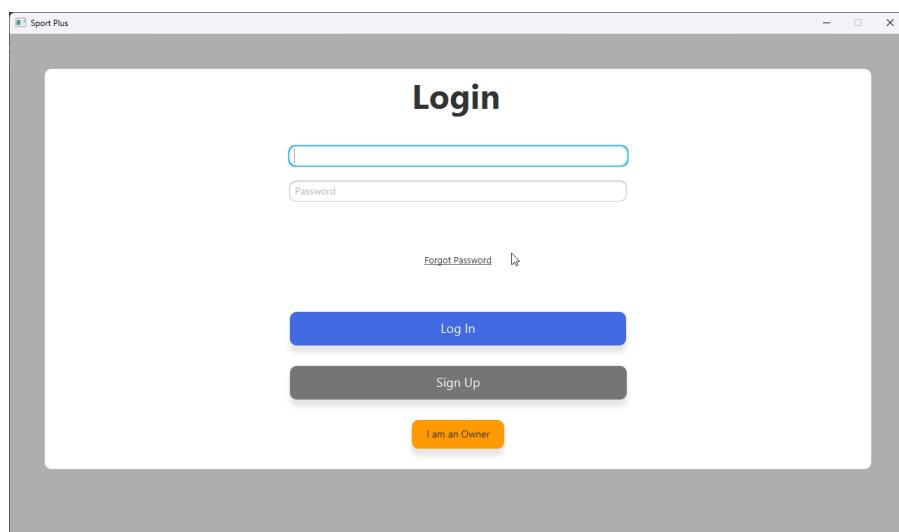


Figura 6: Schermata login lato user.

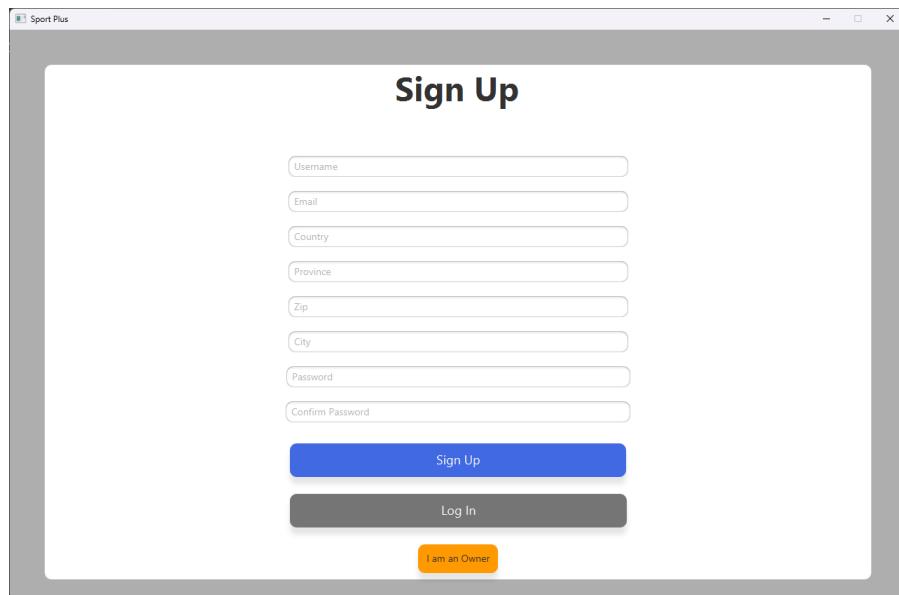


Figura 7: Schermata sign up lato user.

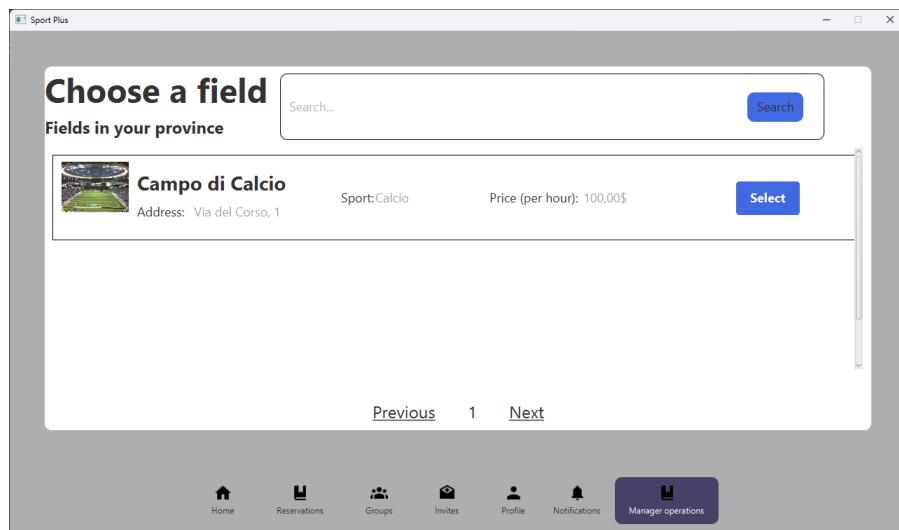


Figura 8: Schermata home lato user.

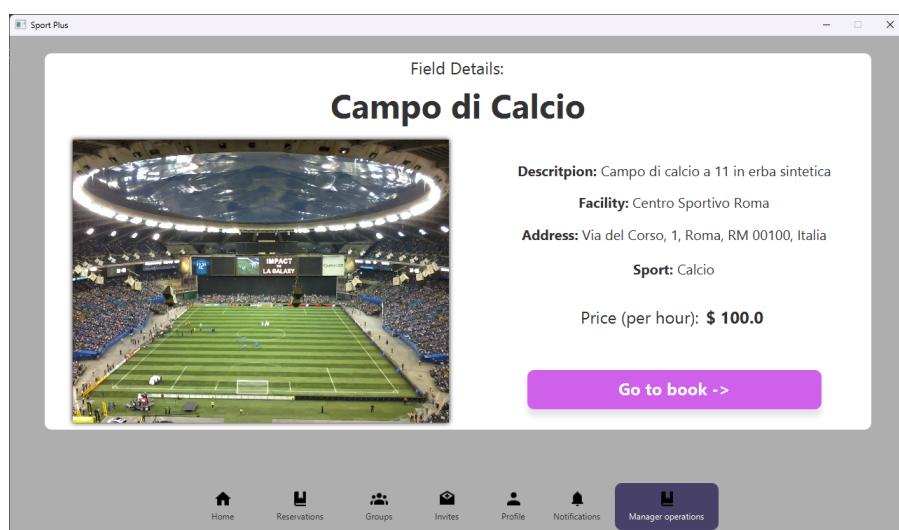


Figura 9: Schermata dettagliata del campo sportivo da prenotare lato user.

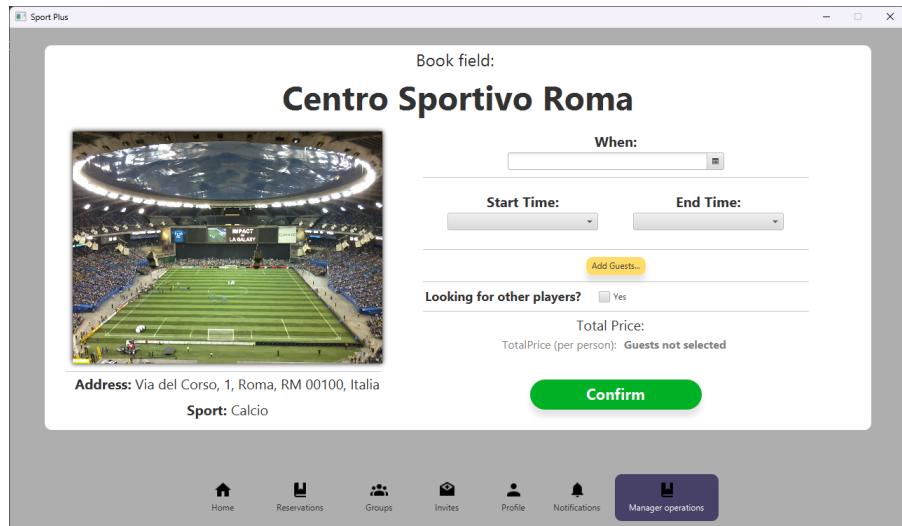


Figura 10: Schermata di prenotazione campo lato user.

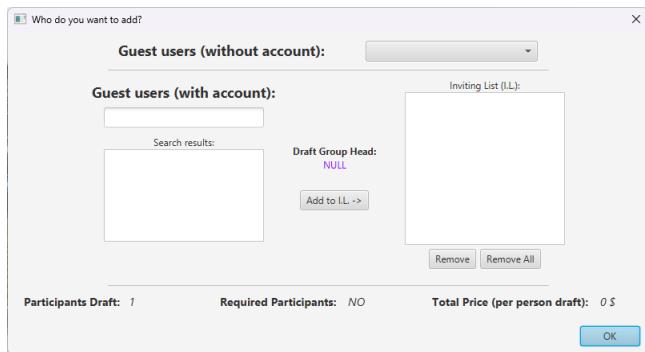


Figura 11: Pop-up di selezione ospiti e invio inviti lato user.

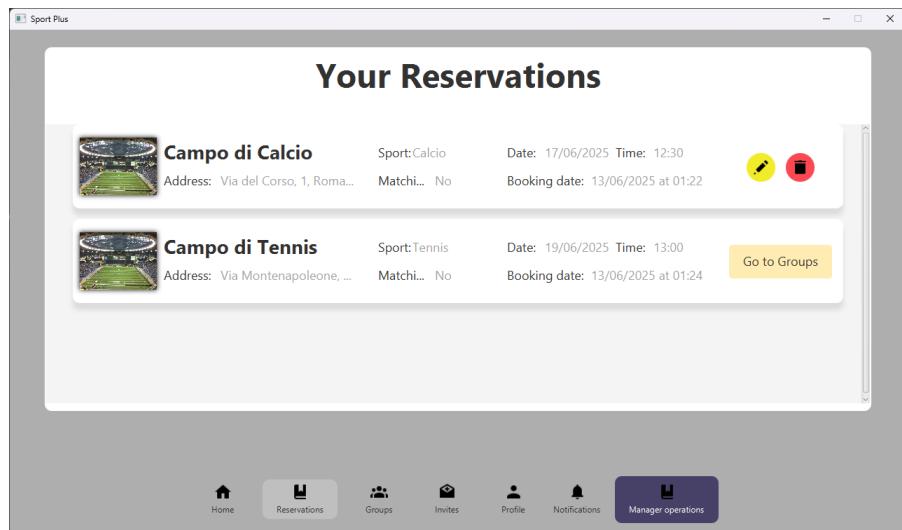


Figura 12: Schermata delle proprie prenotazioni lato user.

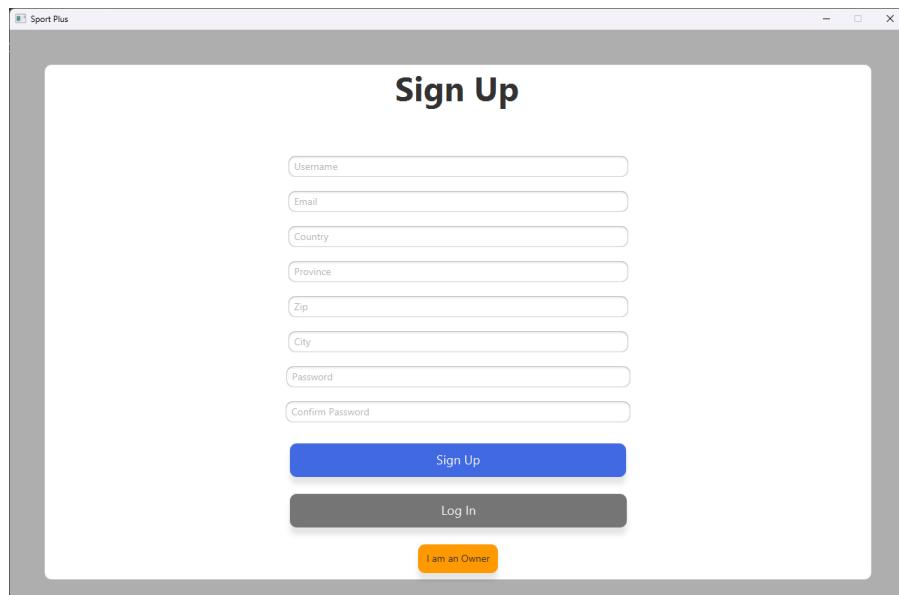


Figura 13: Schermata di modifica prenotazioni lato user. Facendo riferimento ai permessi in Figura 2.

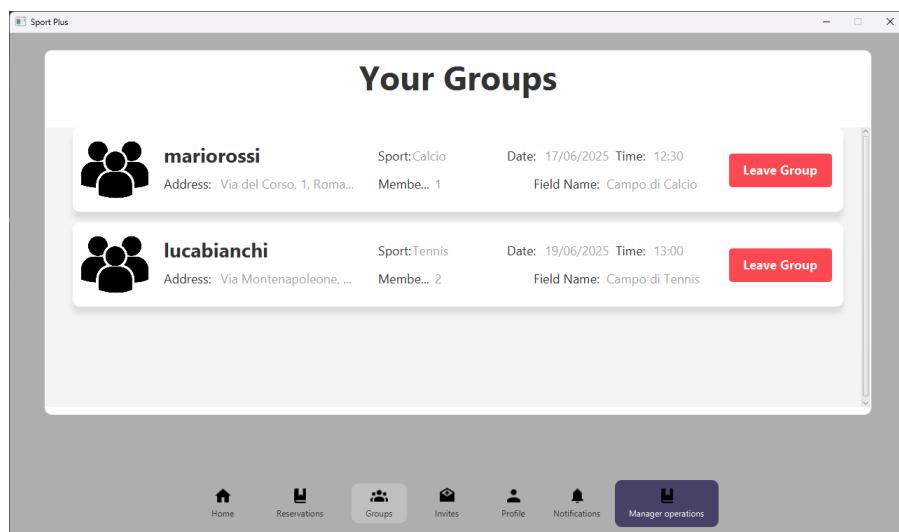


Figura 14: Schermata dei gruppi in cui un utente è membro, lato user.

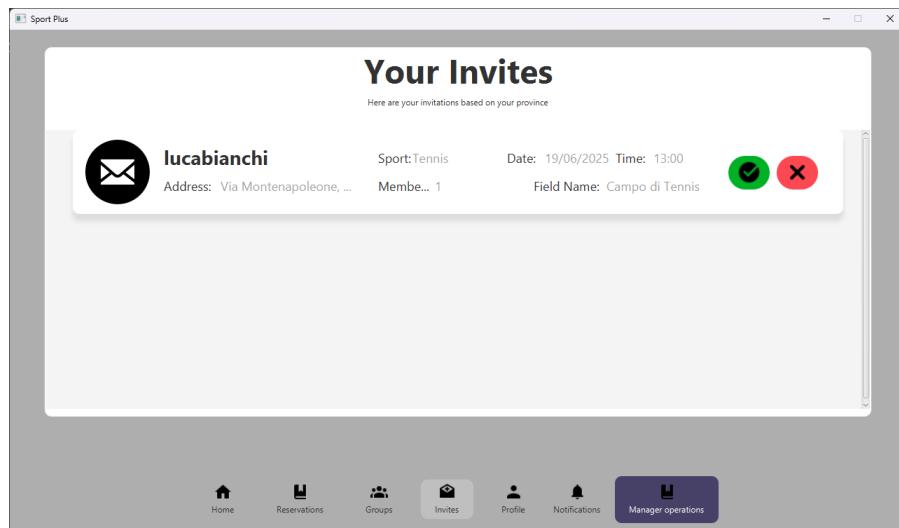


Figura 15: Schermata degli inviti ricevuti lato user.

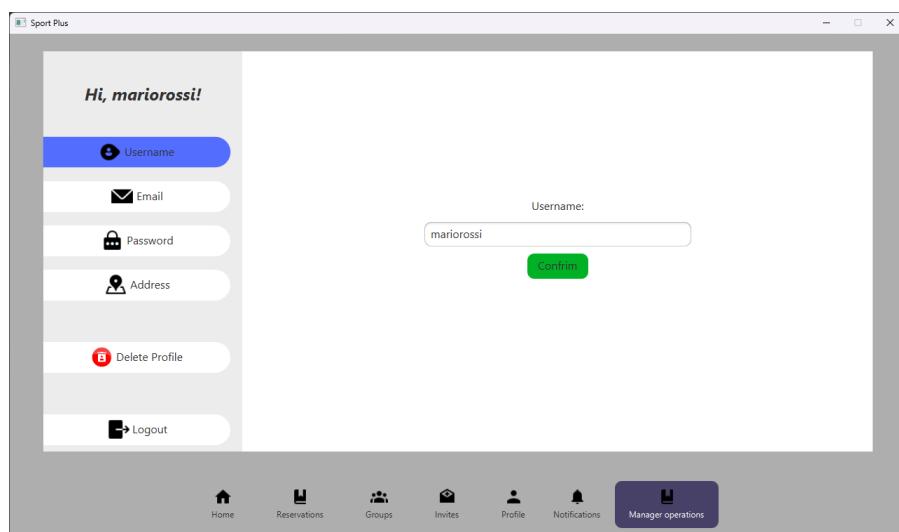


Figura 16: Schermata di modifica username lato user.

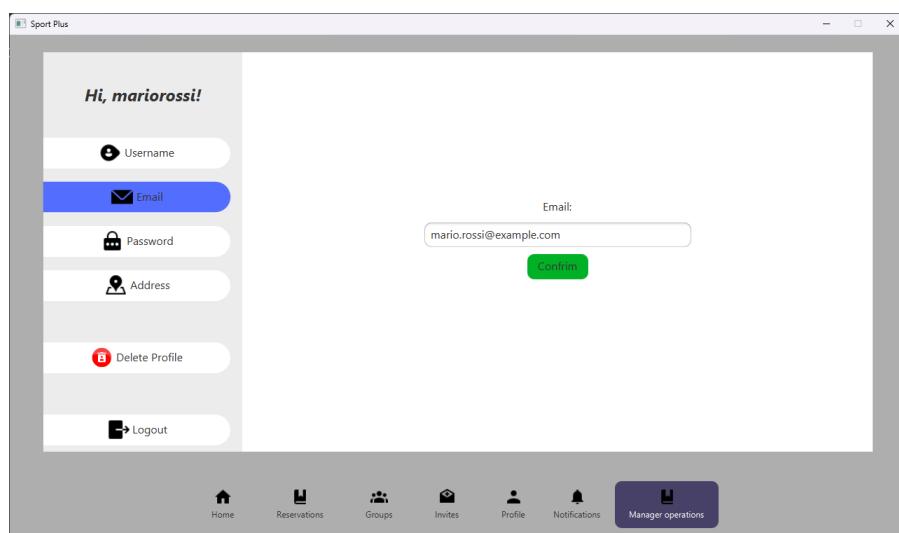


Figura 17: Schermata di modifica email lato user.

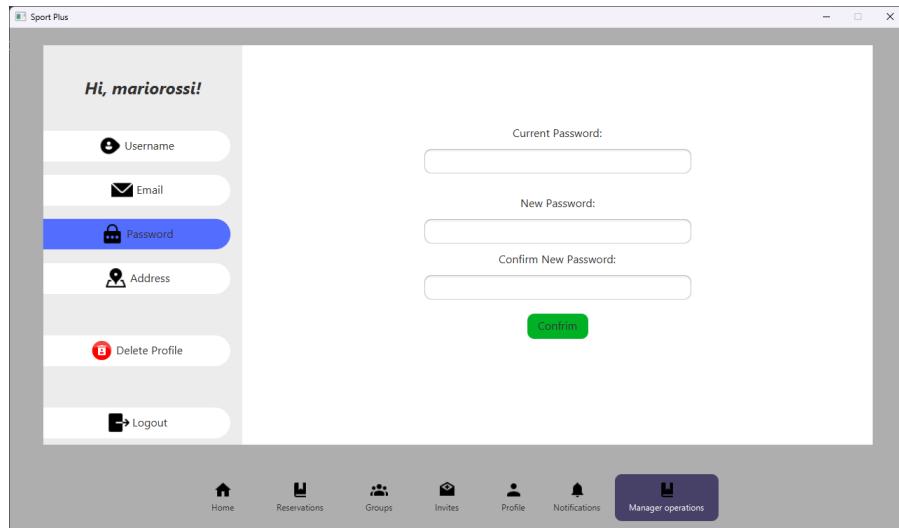


Figura 18: Schermata di modifica password lato user.

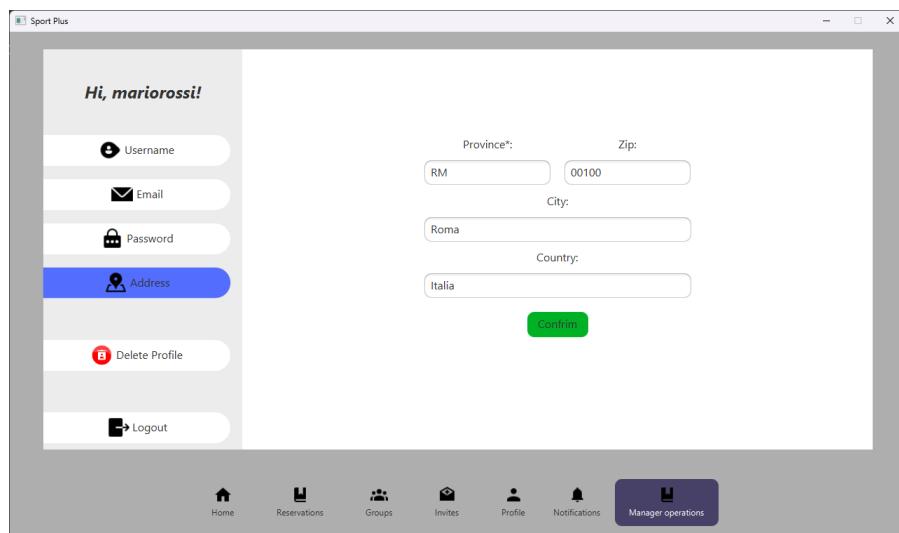


Figura 19: Schermata di modifica indirizzo di residenza lato user.

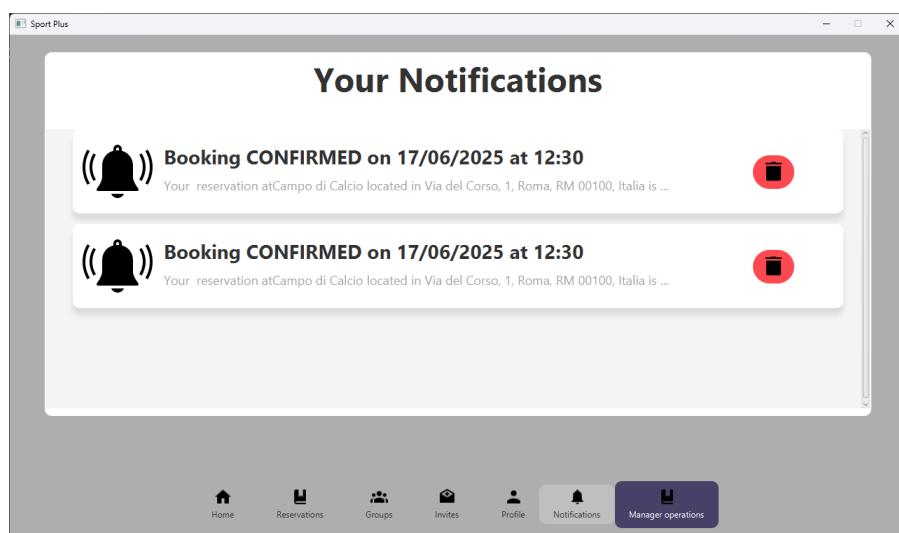


Figura 20: Schermata delle notifiche ricevute lato user.

2.4.2 Manager Side

Alcune schermate lato Manager non sono presenti perché sono analoghe a quelle di **User** o **Owner**, facendo riferimento alla tabella in Figura 2.

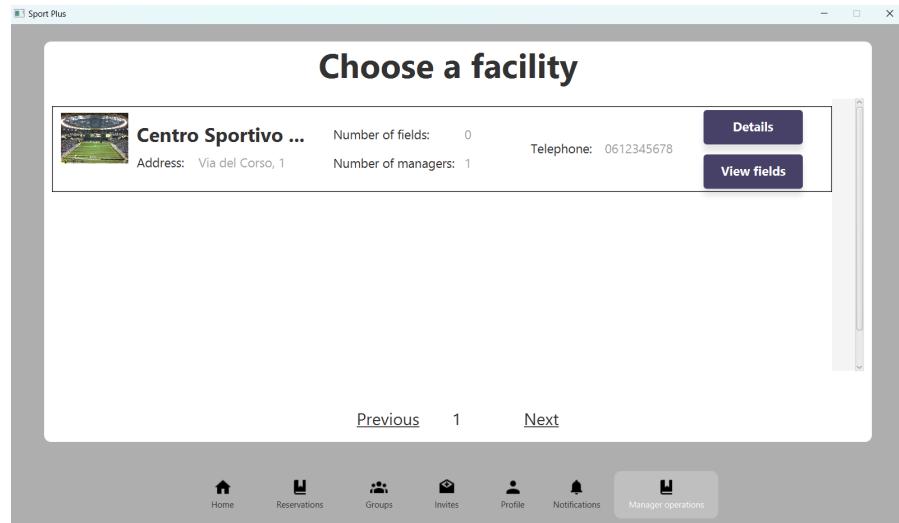


Figura 21: Lista impianti sportivi lato manager.

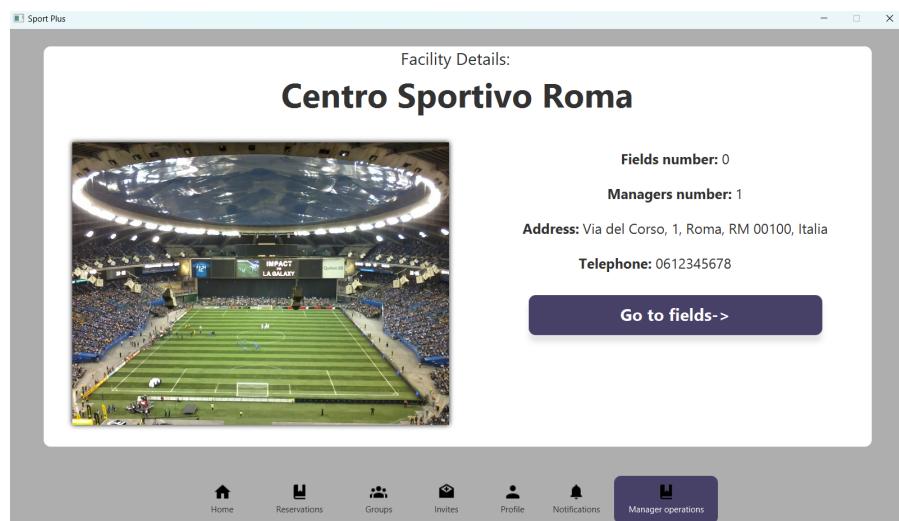


Figura 22: Dettagli impianto sportivo lato manager.

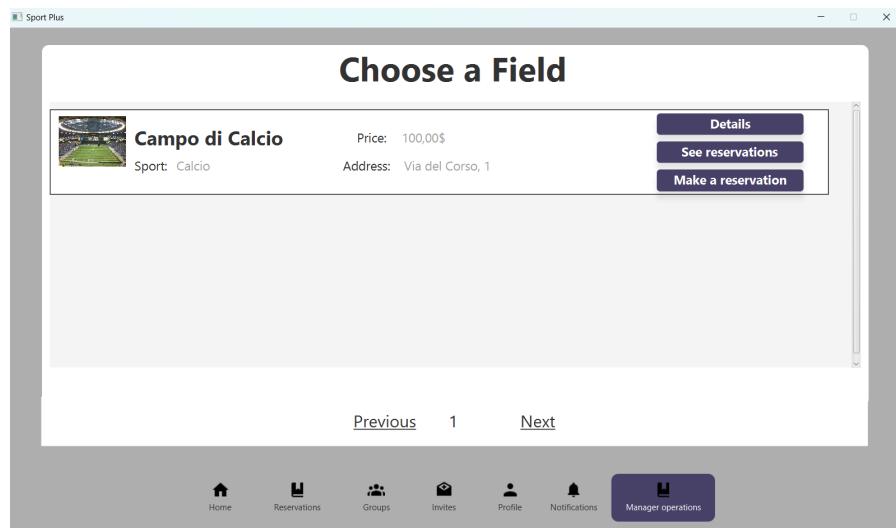


Figura 23: Lista campi sportivi lato manager.

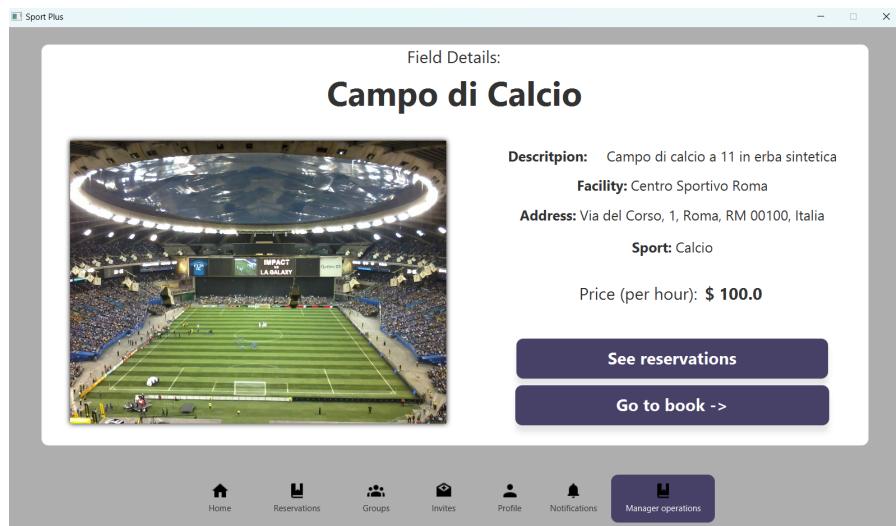


Figura 24: Dettagli campo sportivo lato manager.

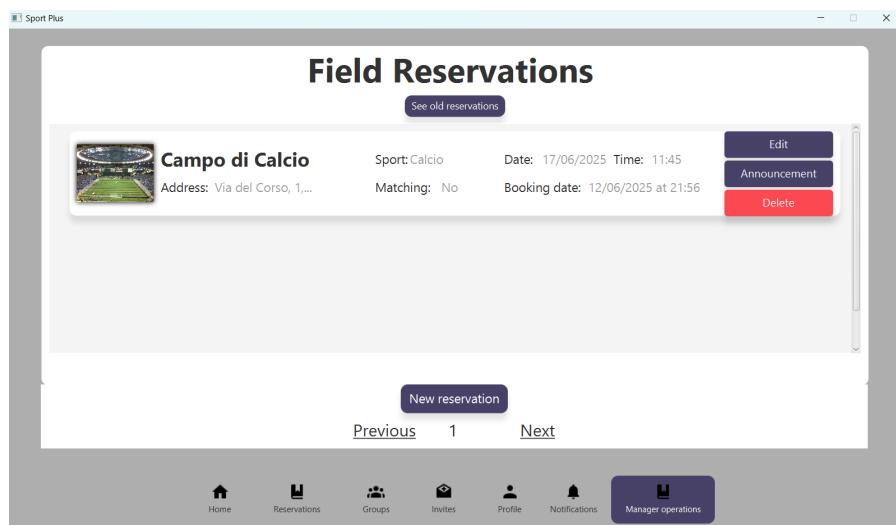


Figura 25: Lista prenotazioni lato manager.

2.4.3 Owner Side

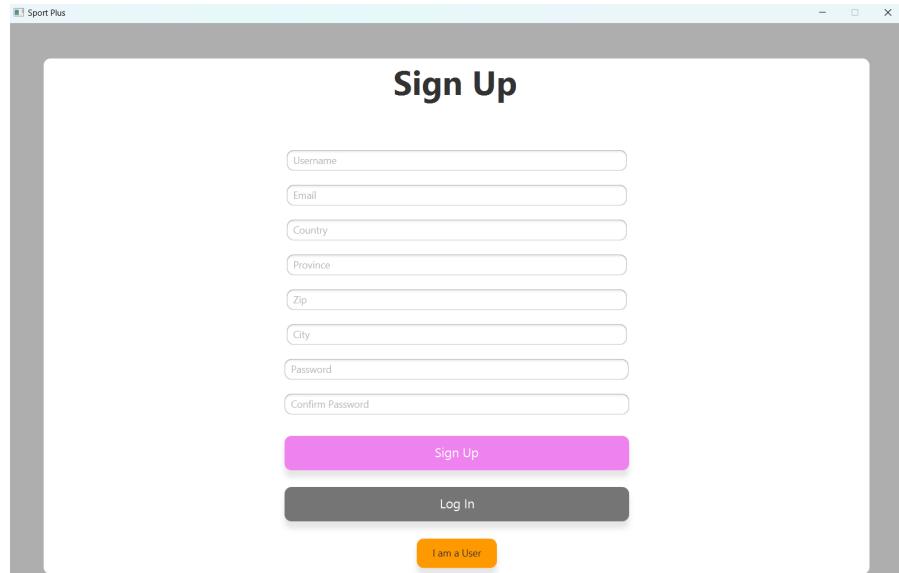


Figura 26: Schermata signUp lato owner.

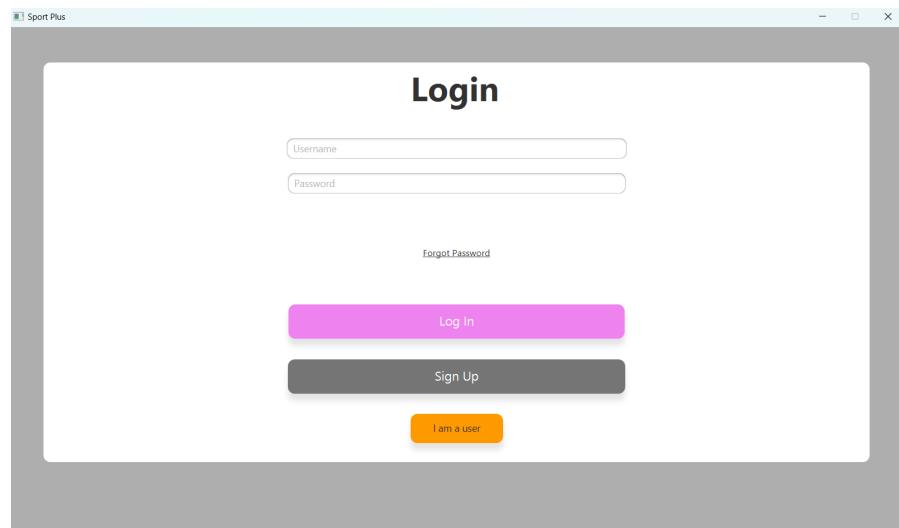


Figura 27: Schermata login lato owner.

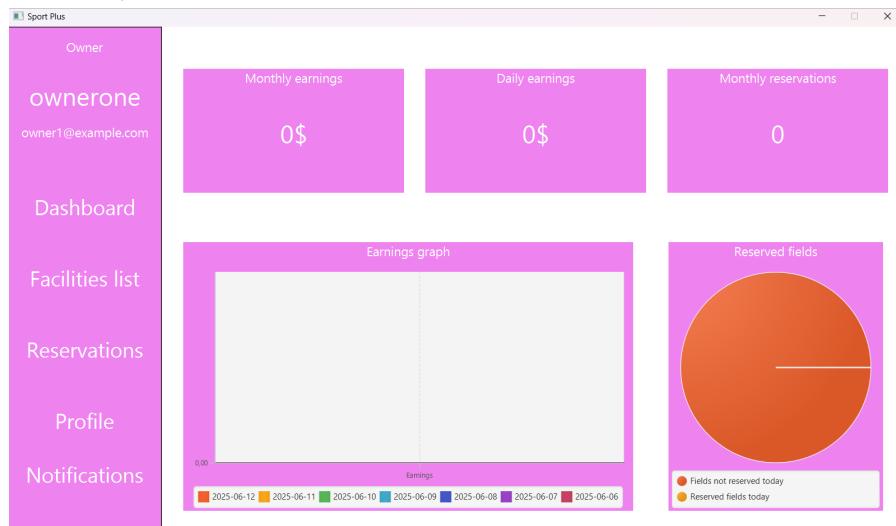


Figura 28: Dashboard Owner.

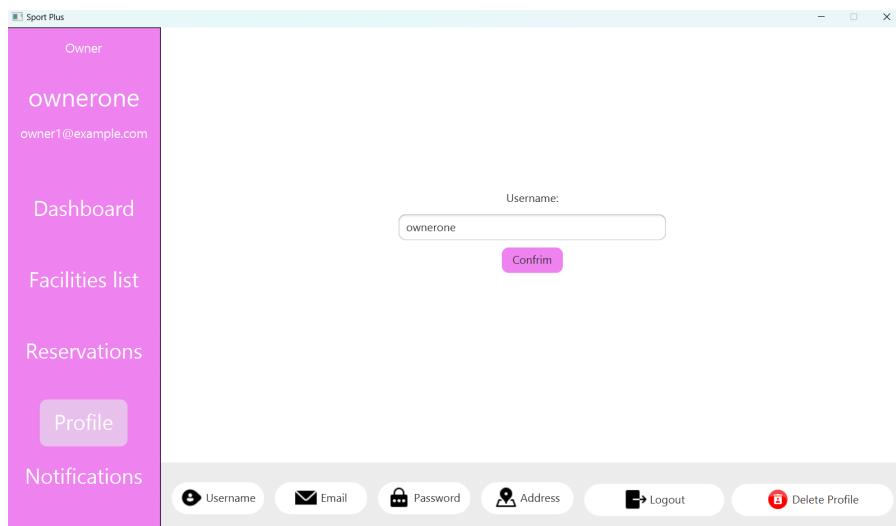


Figura 29: Profilo lato owner.

Figura 30: Schermata creazione impianto sportivo.

Sport Plus

Owner
ownerone
owner1@example.com

Dashboard
Facilities list
Reservations
Profile
Notifications

Name: _____

Price: _____

Description: _____

Sports:

- Calcio
- Tennis
- Pallavolo
- Football

New
Image:
Upload image
Confirm
Add another field

Figura 31: Schermata creazione campo sportivo.

Your Notifications

((Bell)) Booking CONFIRMED on 17/06/2025 at 11:45
Your reservation at Campo di Calcio located in Via del Corso, 1, Roma, RM 00100, Italy... X

Owner
ownerone
owner1@example.com

Dashboard
Facilities list
Reservations
Profile
Notifications

Figura 32: Notifiche owner.

Add managers

Results for 'm'

lucabianchi
Province: MI Email: luca.bianchi@example.com
City: Milano Add

Previous 1 Next

Confirm

Owner
ownerone
owner1@example.com

Dashboard
Facilities list
Reservations
Profile
Notifications

Figura 33: Aggiunta manager.

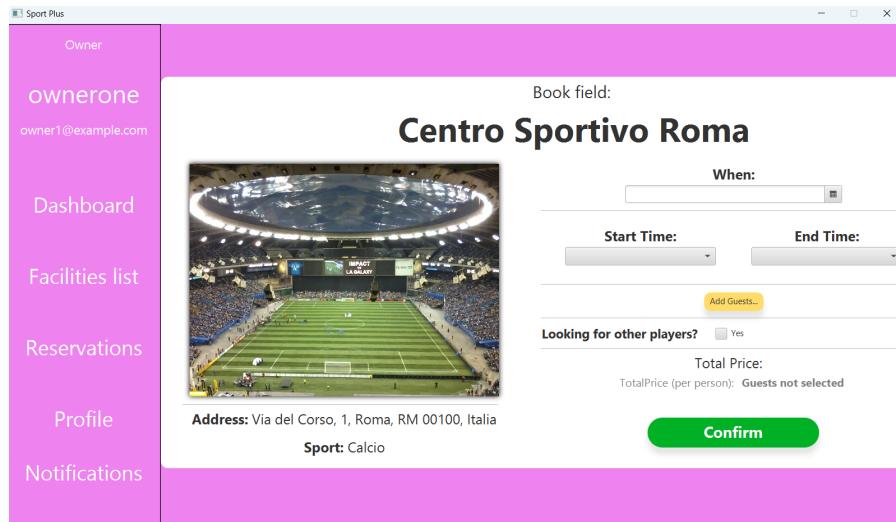


Figura 34: Creazione prenotazione lato owner.

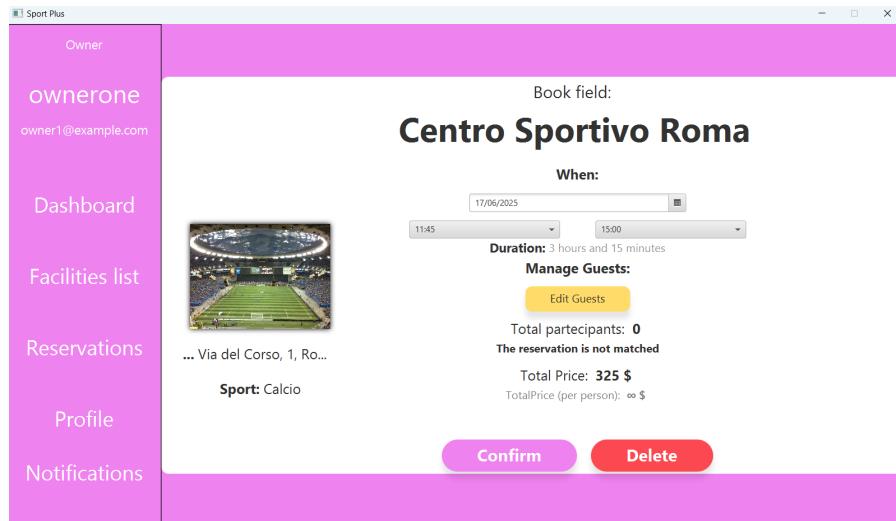


Figura 35: Modifica prenotazione lato owner.

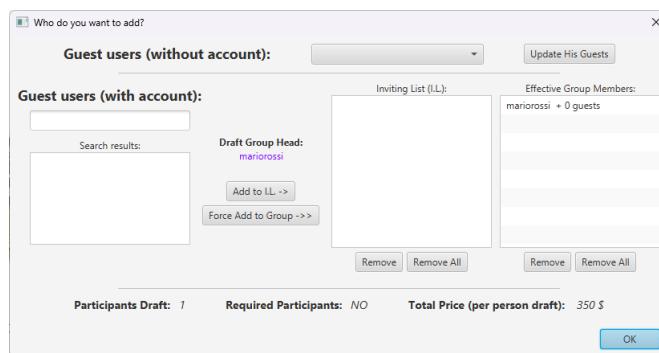


Figura 36: Pop-up di gestione dei membri del gruppo relativo ad una prenotazione lato manager e owner.

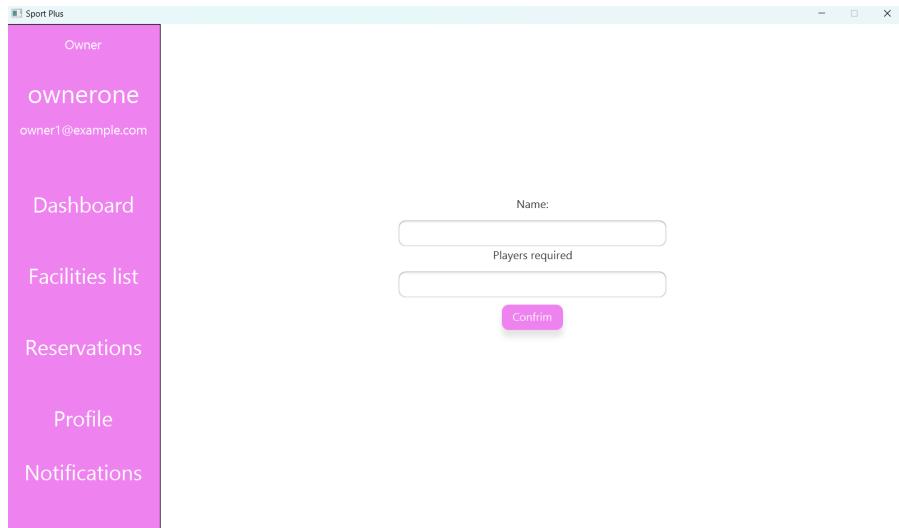


Figura 37: Creazione sport.

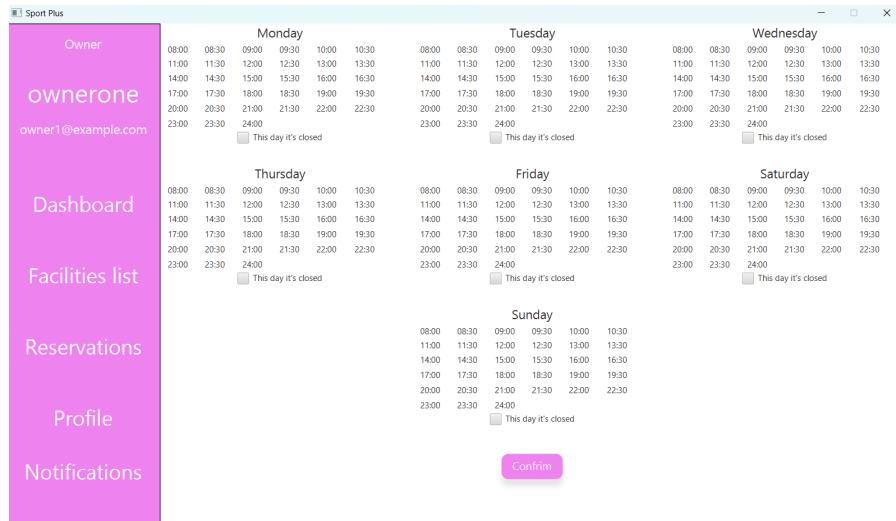


Figura 38: Creazione orari di apertura.

Figura 39: Creazione annuncio lato owner.

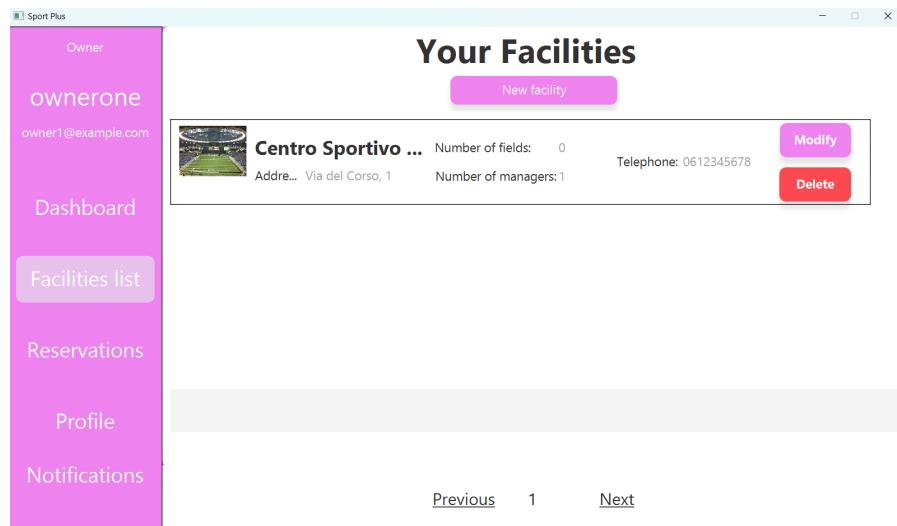


Figura 40: Gestione impianti sportivi.

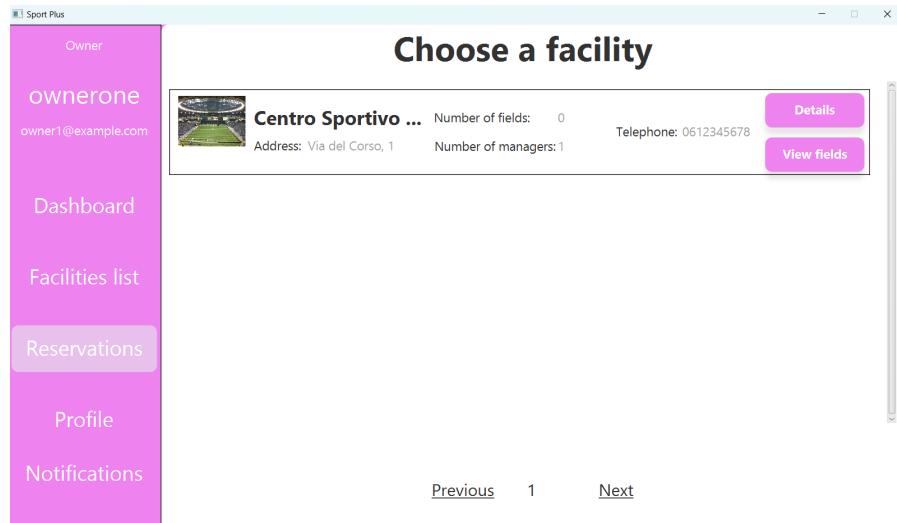


Figura 41: Lista impianti sportivi lato owner.

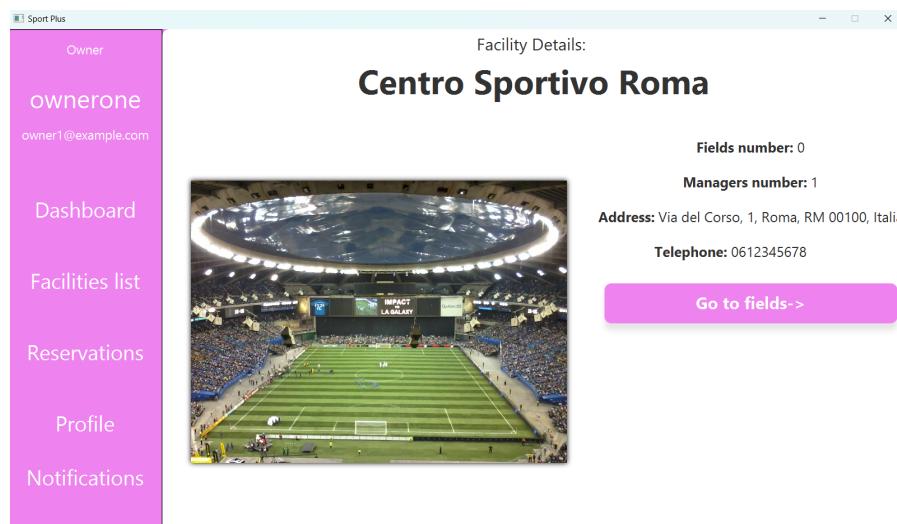


Figura 42: Dettagli impianto sportivo lato owner.

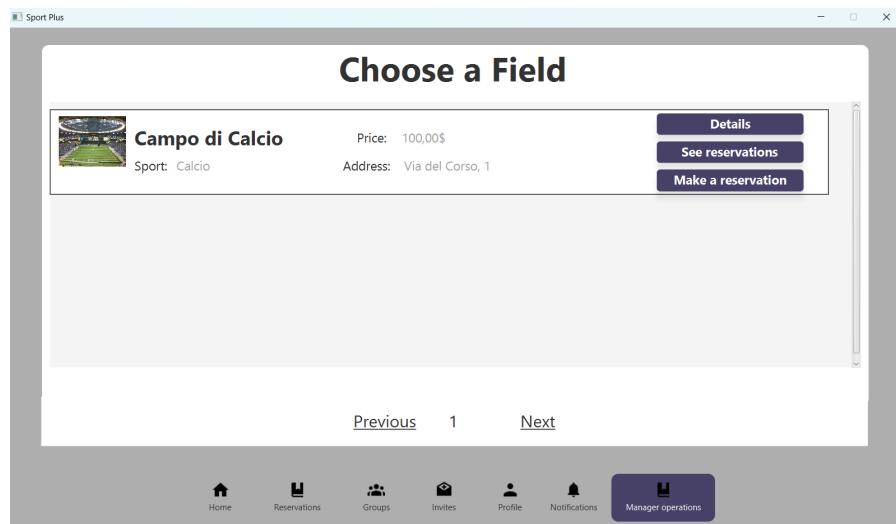


Figura 43: Lista campi sportivi lato manager.

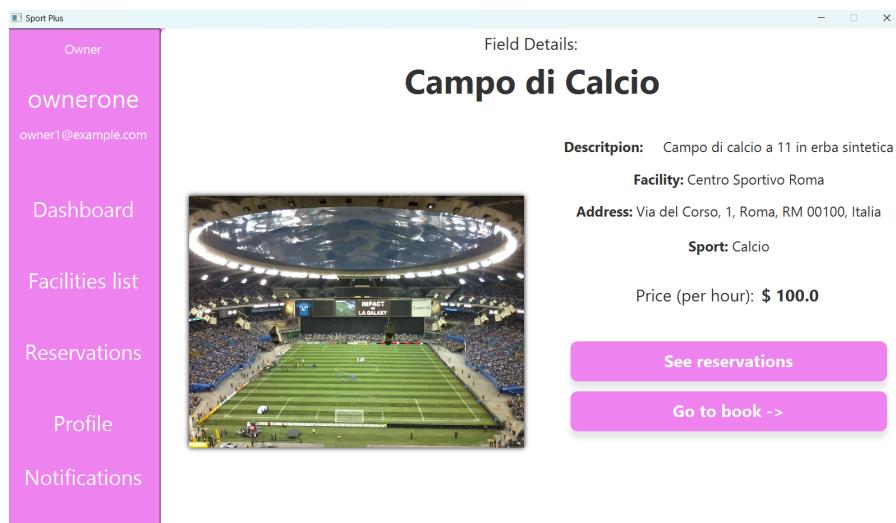


Figura 44: Dettagli campo sportivo lato owner.

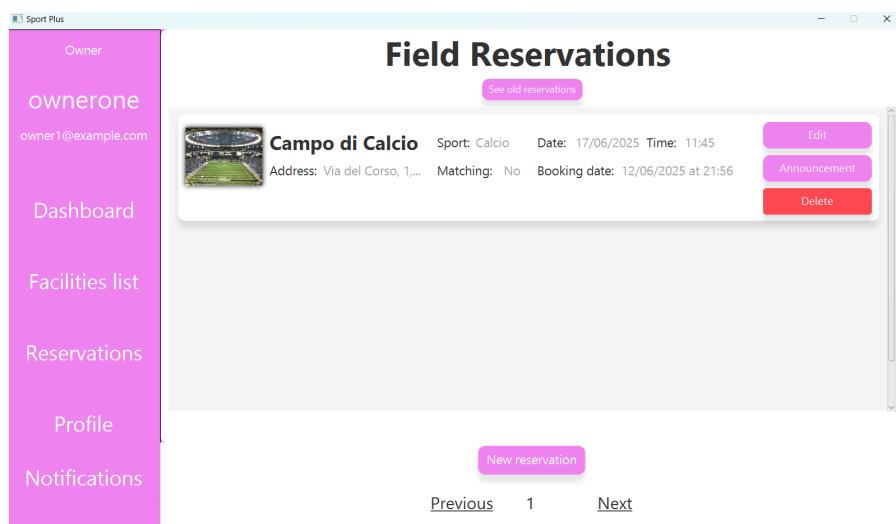


Figura 45: Lista prenotazioni lato owner.

2.5 Navigation Diagrams

In questo paragrafo vengono rappresentate la pagine percorribili da un utente o **Manager** (Figura 46) e quelle percorribili da un **Owner** (Figura 47). I diagrammi mostrati sono in totale due perché sono due percorsi di navigazione **paralleli**, quindi non si incontrano mai, dato che viene effettuato l'accesso anche da GUI e tabelle di DB differenti.

Per semplicità nel disegno, entrambi i diagrammi presentano un parallelogramma ("Menu UI") che non rappresenta una schermata vera e propria, ma simboleggia un **menu interattivo persistente**. Quindi il menu è sempre presente e sempre raggiungibile da ogni schermata la persona si trovi.

Politiche e regole dei Navigation Diagrams:

- **Entrambi** i diagrammi **iniziano** dalla schermata "**Login Page**"
- Le **schermate** effettive navigabili sono rappresentate come dei rettangoli dai bordi arrotondati.
- La direzione ed il verso della freccia indicano il **verso di navigabilità** (ad eccezione del menù che è persistente).
- Alcune schermate sono raggiungibili solamente sotto alcune condizioni, esplicitate sulle frecce.
- Dalle schermate "a vicolo cieco" (senza frecce uscenti) è possibile uscirne solamente tramite il menu persistente.
- La navigazione all'interno del riquadro tratteggiato in **fucsia** (Figura 46) è disponibile solamente se un utente ha i permessi di Manager, ovvero avendo almeno una Facility assegnata.
- In alcune schermate sono presenti anche dei collegamenti ad altre, perché è possibile raggiungerle **anche** in questi modi.

2.5.1 User-Manager's Diagram

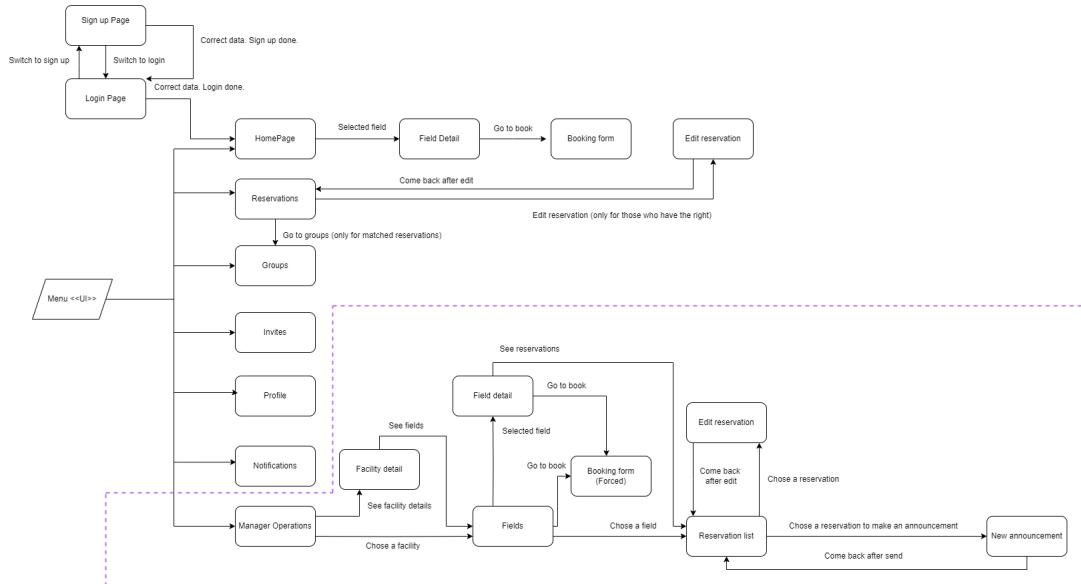


Figura 46: Navigation Diagram relativo alle schermate di navigazione lato utente, con estensione per il gestore (Manager). Il riquadro tratteggiato in **fucsia** è attivo solamente se un utente ha i permessi di Manager, ovvero avendo almeno una Facility assegnata.

Alcune delle schermate più importanti di quelle mostrate in Figura 46 sono:

- **HomePage**: schermata principale raggiungibile dopo il login e dal menù. Mostra i campi relativi alla propria provincia di appartenenza oppure consente di ricercarli secondo le proprie preferenze.
- **Booking Form**: schermata di prenotazione campo. Il Manager ha più permessi rispetto agli utenti, potendo gestire i membri del gruppo. Per uscirne è necessario usare il menù.

- **Edit Reservaiton:** schermata di modifica di una Reservation. Il Manager ha più permessi rispetto agli utenti, potendo gestire i membri del gruppo (Figura 2). Per uscirne è necessario usare il menù.
- **Reservations:** schermata che racchiude tutte le proprie Reservation.
- **Profile:** schermata che consente di visualizzare e modificare le informazioni dell'account.
- **Manager Operations:** schermata disponibile **soltamente** ai Manager. Consente di gestire gli impianti (Facility) assegnati dagli Owner. Le operazioni disponibili sono mostrate in Figura 2.
- **New Announcement:** schermata disponibile **soltamente** ai **Manager**. Consente di inviare un annuncio a tutti i membri di una prenotazione.

2.5.2 Owner's Diagram

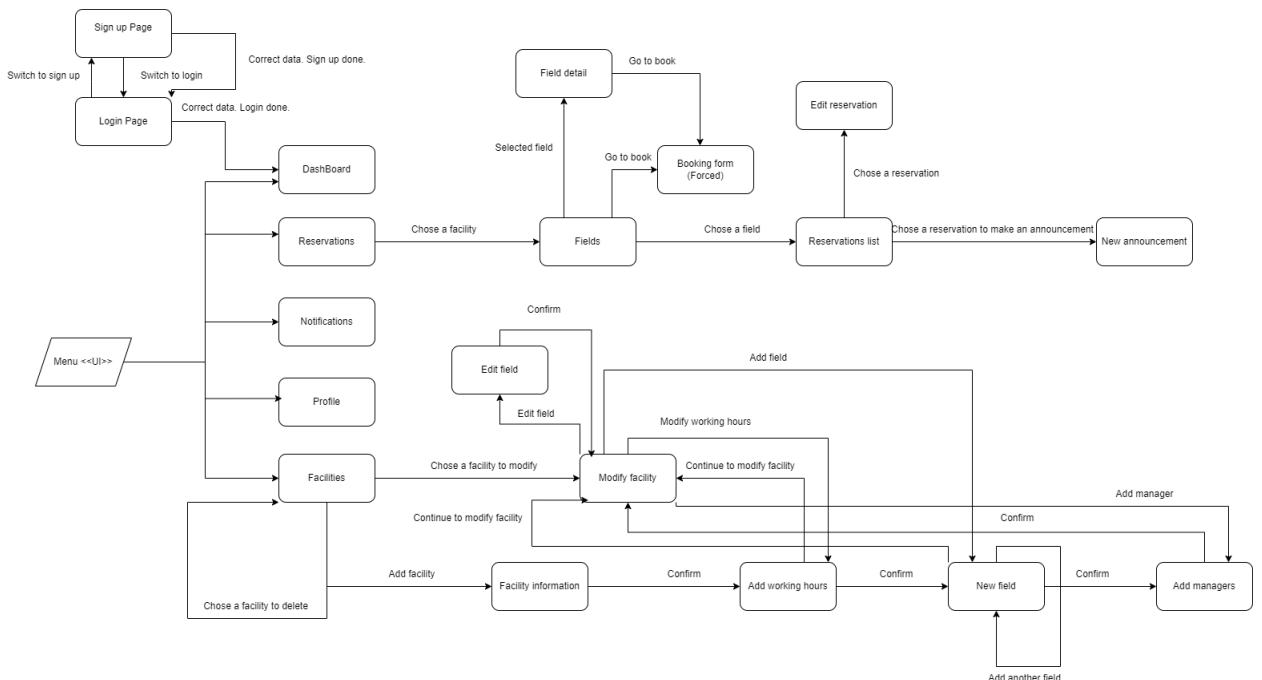


Figura 47: Navigation Diagram relativo alle schermate di navigazione lato **Owner**.

Alcune delle schermate più importanti di quelle mostrate in Figura 47 sono:

- **HomePage:** schermata principale raggiungibile dopo il login e dal menù. Mostra i dati statistici di possibile interesse al proprietario riguardanti i suoi impianti sportivi.
- **Booking Form:** schermata di prenotazione campo. In questo caso con più permessi rispetto all'utente normale (Figura 2). Per uscirne è necessario usare il menù.
- **Edit Reservaiton:** schermata di modifica di una Reservation. Anche in questo caso con il proprietario ha più permessi di un normale utente potendo gestire i membri del gruppo (Figura 2). Per uscirne è necessario usare il menù.
- **Reservations list:** schermata che racchiude tutte le Reservation dell'impianto scelto.
- **Profile:** schermata che consente di visualizzare e modificare le informazioni dell'account.
- **New facility:** schermata di creazione di un impianto(Figura 2).
- **Modify facility:** schermata di modifica e gestione di un impianto(Figura 2).
- **New field:** schermata di creazione di un campo(Figura 2).
- **Modify field:** schermata di modifica e gestione di un campo(Figura 2).

- **Add managers:** schermata in cui il proprietario può aggiungere dei Manager all’impianto(Figura 2).
 - **Facilities:** schermata in cui è possibile selezionare un impianto per vederne i dettagli o cancellarlo (Figura 2)..
 - **New Announcement:** Consente di inviare un annuncio a tutti i membri di una prenotazione.

2.6 Class Diagrams

Nei seguenti diagrammi di *get* e *set* non vengono mostrati per semplificare la visibilità. Vengono mostrati solamente quelli che hanno maggiore rilevanza, magari chiamando al loro interno altri metodi, come ad esempio **setConfirmed(...)** di **Reservation** del DomainModel.

Il modo in cui interagiscono i vari packages è stato mostrato in Figura 1.

Nei diagrammi se vengono mostrati dei puntini (...) significa che sono presenti altri parametri, in particolare ArrayList.

2.6.1 DomainModel

Come mostrato nel diagramma in Figura 48, viene implementato il design pattern *Observer*, utile in questo caso per far passare la flag **isConfirmed** a **true, solamente** per le prenotazioni **matched**. L'*Observer* viene innescato quando un gruppo viene **riempito**, così da considerare la prenotazione confermata rendendo la prenotazione effettiva. In questo caso la classe **NotificationController** della *BusinessLogic*, che si occupa di inviare le notifiche, **implementa l'Observer**, come mostrato in Figura 49. Questo viene mostrato più in dettaglio al paragrafo 3.2.9. Nonostante ciò, in questo sistema, l'utilizzo dell'*observer* diventa particolarmente utile per la scalabilità del progetto, ovvero per rendere più agevoli le estensioni o aggiornamenti futuri.

L'attributo **target** di **Person** indica il tipo di persona modo rapido come ad esempio "*User*".

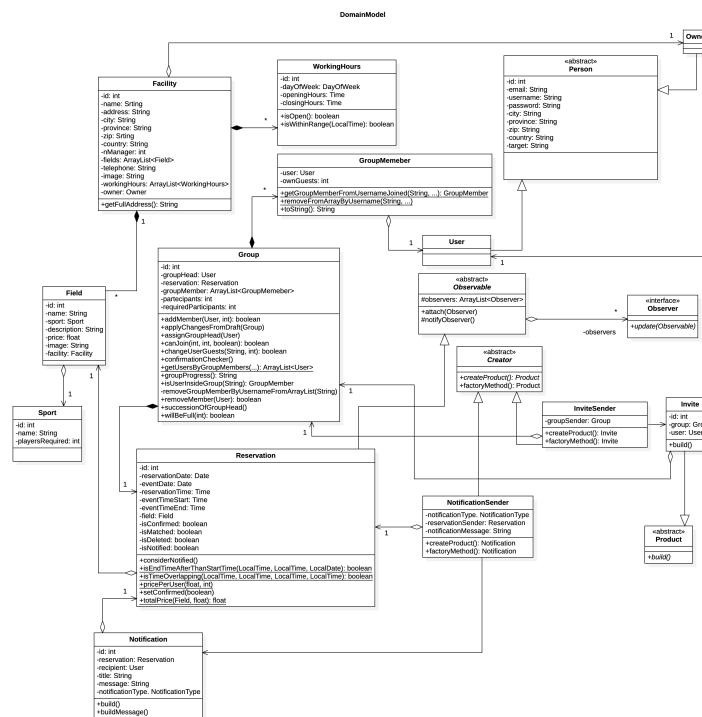


Figura 48: Diagramma UML del package DomainModel.

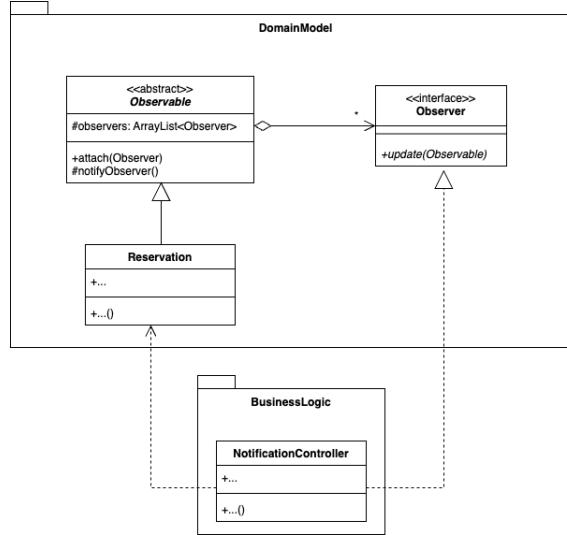


Figura 49: Esempio di interazione tra l’interfaccia **Observer** e l’observer concreto (**NotificationController**).

2.6.2 BusinessLogic

Come mostrato nel diagramma della Figura 50, molte classi della BusinessLogic hanno come campi i DAO del package ORM. Questo è utile per i test, come spiegato al paragrafo 4.2. Quindi la corretta interazione dei package viene mostrata in Figura 1.

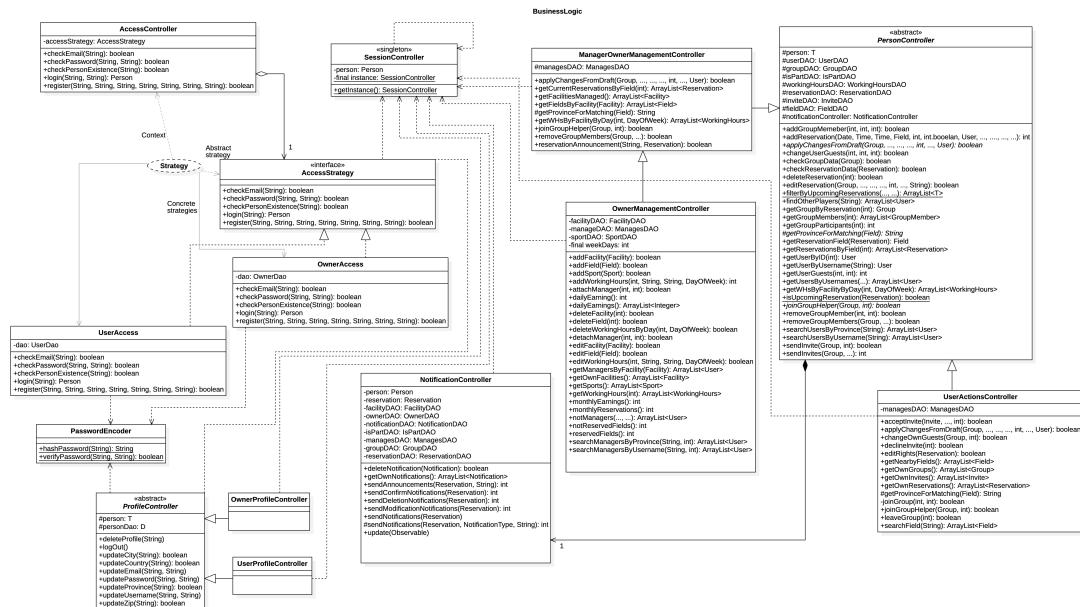


Figura 50: Diagramma UML del package BusinessLogic.

2.6.3 ORM

Come mostrato nel diagramma della Figura 51, è presente un design pattern **singleton** nella classe **ConnectionManager** che si occupa di gestire la connessione con il database **PosgreSQL**. Invece la classe **ConnectionHolder** è una classe astratta dalla quale tutti i DAO ereditano, così da gestire la connessione in modo centralizzato, interfacciandosi anche con il *singleton*.

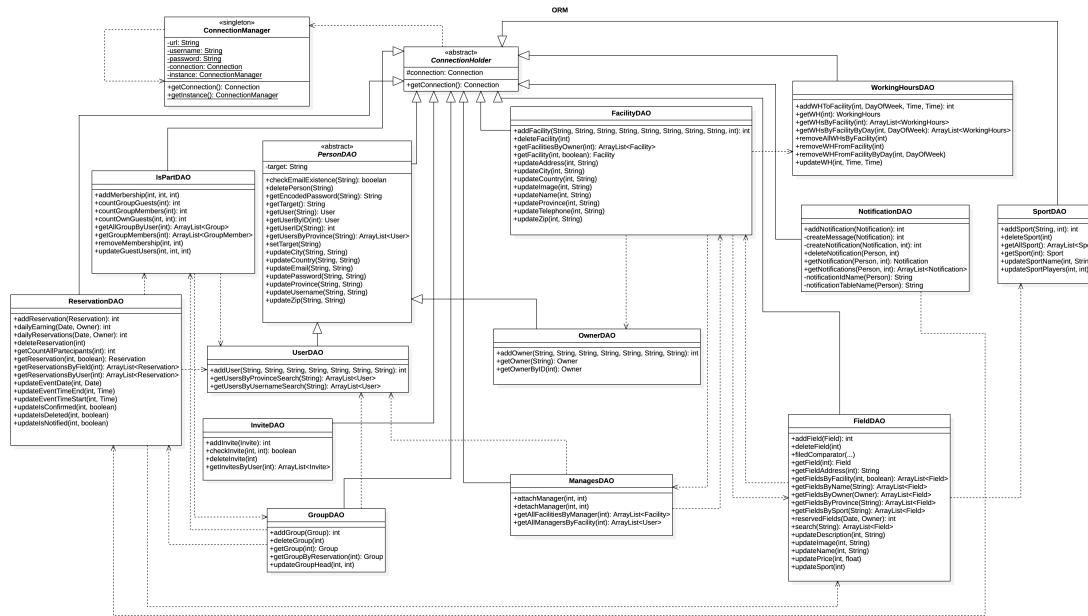


Figura 51: Diagramma UML del package ORM.

2.6.4 GUI

Nel diagramma in Figura 52 vengono mostrati solo le generalizzazioni ed i metodi delle classi di GUI dato che sono molto voluminose. Infatti a differenza degli altri diagrammi, la GUI non presenta un’architettura particolare dato che ogni **GUI Control** corrisponde ad una schermata **FXML**.

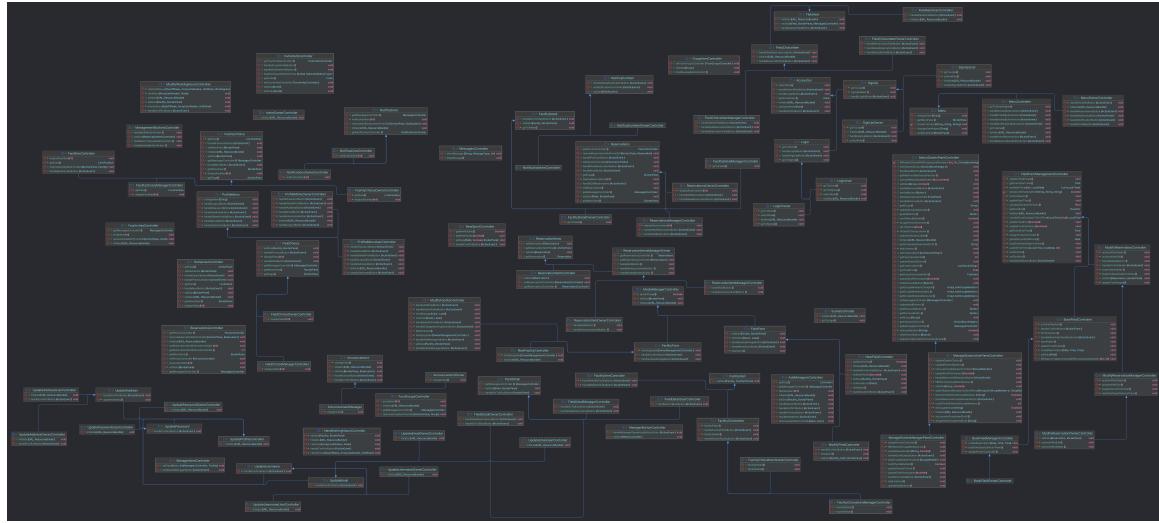


Figura 52: Diagramma UML del package GUIControl.

2.7 Database

Per quanto riguarda il database si mostra il modello relazionale(Figura 54) e il modello ER(figure 53). Ci sono state varie scelte progettuali come quella di realizzare due tabelle distinte per le notifiche **Owner** e quelle **User** in modo da semplificarne la gestione mentre invece il contenuto dei tuoi tipi di notifica viene unificato nella tabella **Message** per ridurre la ridondanza. Le tabelle definite sono:

- **User** rappresenta l'entità **User** e **Manager**.
 - **Owner** rappresenta l'entità **Owner**.
 - **Reservation** rappresenta l'entità **Reservation** e risolve la relazione **at**.
 - **Group** rappresenta l'entità **Group** e risolve le relazioni **Regarding** e **Group head**. Tenendo conto che un gruppo viene sempre creato anche se all'interno c'e una sola persona, dato che gli viene associata una **Reservation**.

- **Invite** rappresenta l'entità **Invite** e risolve le relazioni **From** e **To**.
- **IsPart** risolve la relazione **Is part** tra **User** e **Group**.
- **Manages** risolve la relazione **Manages** tra **User** e **Facility**.
- **Sport** rappresenta l'entità **Sport**.
- **Message** rappresenta un messaggio personalizzato che viene creato e salvato (per evitare ridondanze) quando si invia un annuncio.
- **WH** rappresenta l'entità **WorkingHours** e risolve la relazione **Is Open**. In particolare rappresento uno slot orario continuo in cui l'impianto è aperto.
- **Facility** rappresenta l'entità **Facility** e risolve la relazione **Owns**.
- **Field** rappresenta l'entità **Field** e risolve le relazioni **Include** e **Used for**.
- **NotificationUser** rappresenta l'entità **Notification** per quanto riguarda gli **User** (o **Manager**) e risolve le relazioni **Content**, **About** e **To**.
- **NotificationOwner** rappresenta l'entità **Notification** per quanto riguarda gli **Owner** e risolve le relazioni **Content**, **About** e **To**.

2.7.1 Modello ER

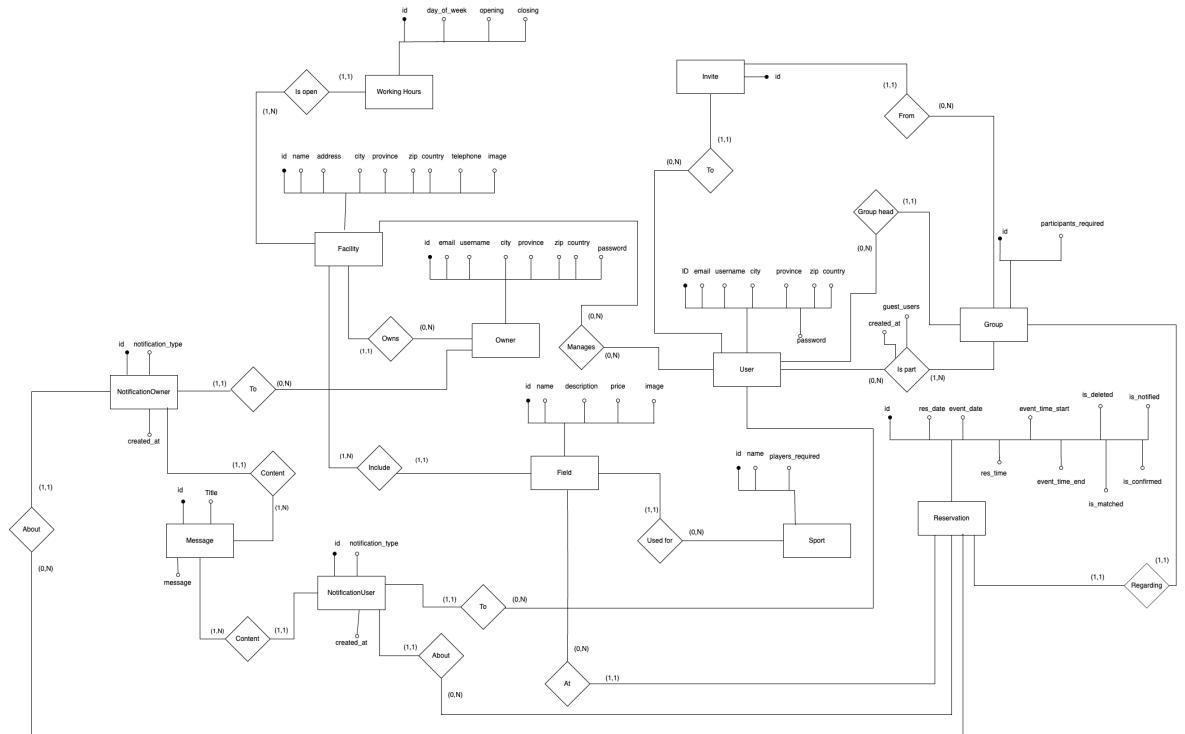


Figura 53: Modello ER relativo al database.

2.7.2 Modello Relazionale

```
Facility(id, name, address, city, province, zip, country, telephone, image, id_owner)

Owner(id, email, username, city, province, zip, country, password)

Field(id, name, id_sport, description, price, image, id_facility)

User(id, email, username, city, province, zip, country, password)

Manages(id_facility, id_user)

Sport(id, name, players_required)

Group(id, group_head, participants_required, id_reservation)

Is Part(id_user, id_group, guest_users, created_at)

Reservation(id, res_date, event_date, res_time, event_time_start, event_time_end, id_field,
is_confirmed, is_matched, is_deleted, is_notified)

Invite(id, id_group, id_user)

WH(id, day_of_week, opening, closing, id_facility)

NotificationUser(id, id_user, notification_type, id_message, id_reservation, created_at)

NotificationOwner(id, id_owner, notification_type, id_message, id_reservation, created_at)

Message(id, title, message)
```

Figura 54: Modello Relazionale relativo al database.

3 Implementazione

3.1 Domain Model

In questo package sono implementate le classi per le entità di sistema ed altre utili ai design pattern. Le classi corrispondenti alle tabelle del database hanno due costruttori overload: uno per la lettura (creazione di un oggetto da un database e immagazzinamento tramite DomainModel) e uno per la scrittura (creazione di un oggetto da un DomainModel e salvataggio su database). Viene mostrato un esempio nello Snippet 1. Le classi che rappresentano entità di sistema (come **Facility**, **Group**, ecc.) implementano la *marker interface* **Serializable** per abilitare un oggetto ad essere convertito in una sequenza di byte, in questo caso utile per effettuare rapide deep copy , data la presenza di numerose dipendenze di oggetti a catena.

3.1.1 GroupMember

Questa classe è una classe finale e funge da struct per rappresentare un membro del gruppo con i relativi ospiti associati. Quindi come campi ha:

- **User**: utente membro del gruppo;
- **OwnGuests**: numero degli ospiti (senza account) associati alla sua presenza.

3.1.2 Group

Questa classe rappresenta l'entità Group del database. I campi della classe sono:

- **Id**: id nel database;
- **GroupHead**: capo gruppo.
- **Reservation**: prenotazione associata al gruppo.
- **GuestsUsers**: numero totale di ospiti (senza account) presenti nel gruppo.
- **Participants**: numero dei partecipanti attuali presenti nel gruppo.
- **GroupMemebers**: array di tutti i partecipanti del gruppo con i relativi ospiti associati.
- **RequiredParticipants**: numero dei partecipanti richiesti (massimi) in caso di prenotazione di tipo matched.

```
1 //from DB to DM
2 public Group(int id, User groupHead, Reservation reservation, int
3 requiredParticipants, ArrayList<GroupMember> groupMembers, int participants)
4 {
5     this.id = id;
6     this.groupHead = groupHead;
7     this.reservation = reservation;
8     this.requiredParticipants = reservation.isMatched() ?
9 requiredParticipants : 0;
10    this.groupMembers = groupMembers;
11    this.participants = participants;
12 }
13
14 //from DM to DB
15 public Group(User groupHead, Reservation reservation, int
16 requiredParticipants,int guests) {
17     this.groupHead = groupHead;
18     this.reservation = reservation;
19     this.requiredParticipants = reservation.isMatched() ?
20 requiredParticipants : 0;
21     this.groupMembers = new ArrayList<>();
22
23     if (guests < 0)
24         guests = 0;
25
26     if (groupHead != null) {
27         groupMembers.add(new GroupMember(groupHead, guests));
28         this.participants = guests + 1;
29     }
30     else
```

```

26         this.participants = 0;
27     }

```

Snippet 1: Esempio dei due overload dei costruttori della classe **Group** del DomainModel

3.1.3 NotificationType

NotificationType è un'enumerazione che rappresenta i diversi tipi di notifica. Utile per intraprendere azioni specifiche in base al tipo.

I vari tipi di notifica sono:

- **Confirmation**: notifica di conferma;
- **Modification**: notifica di modifica;
- **Deletion**: notifica di eliminazione;
- **Announcement**: notifica di annuncio personalizzato;
- **Unknown**: tipo sconosciuto, caso di default;

3.1.4 Notification

Questa classe rappresenta l'entità **NotificationUser** e **NotificationOwner** del database. Infatti sul database sono presenti due tabelle diverse (quindi ridondanti) per ridurre la complessità computazionale. Però a livello di DomainModel possiamo avere un'unica classe per entrambe, dato che sono speculari.

I campi della classe sono:

- **Id**: id nel database;
- **Recipient**: persona destinatario.
- **Reservation**: prenotazione associata alla notifica.
- **Title**: titolo della notifica.
- **Message**: messaggio o descrizione della notifica.
- **NotificationType**: tipo (categoria) della notifica identificata tramite enumerazione.

con il metodo **buildMessage()** viene assemblato il messaggio da mostrare al destinatario in base al tipo di notifica e ad alcuni dati. In questo modo vengono riempiti i campi **title** e **message**. L'unica eccezione si ha nella notifica di tipo **Announcement** che ha titolo e messaggio personalizzati, ovvero digitati ad input dalla persona.

3.1.5 Observer

L'interfaccia Observer viene usata per l'implementazione del design pattern *Observer* di tipo pull. Presenta il metodo:

- **update(...)**: metodo di tipo pull. Si occupa di effettuare l'aggiornamento quando viene chiamato il ciclo di notifica facendo un down-cast e richiedendo i dati all'**Observable** effettuando gli opportuni aggiornamenti.

3.1.6 Observable

Questa classe astratta rappresenta l'**Observable** del del design pattern *Observer*, ossia colui che è "osservato". Ha un solo campo ovvero l'array degli observer, **observers**. Come metodi ha:

- **attach(..)**: metodo per aggiungere un observer alla propria lista.
- **notifyObserver()**: metodo usato innescare un ciclo di notifica degli di tutti i propri observer, chiamando i relativi metodi **update(...)**.

3.1.7 Reservation

Questa classe rappresenta un'entità **Reservation**, ovvero una prenotazione effettuata da una persona. Tale prenotazione sarà assegnata ad un unico gruppo. I suoi campi sono:

- **Id**: l'id nel database;
- **ReservationDate**: la data di effettuazione della prenotazione.
- **ReservationTime**: l'ora di effettuazione della prenotazione.
- **EventDate**: la data della prenotazione del campo (evento).
- **EventTimeEnd**: l'ora di fine della prenotazione del campo (evento).
- **EventTimeStart**: l'ora di inizio della prenotazione del campo (evento).
- **Field**: il campo prenotato.
- **IsMatched**: flag che indica se la prenotazione è matched e non matched.
- **IsConfirmed**: flag che indica se la prenotazione è confermata. Se la prenotazione è **non matched**, la prenotazione sarà automaticamente confermata. Invece se la prenotazione è **matched**, sarà confermata quando sarà raggiunto il numero di partecipanti richiesto.
- **IsDeleted**: flag che indica se la prenotazione è stata eliminata.
- **IsNotified**: flag che indica se la notifica di prenotazione è stata inviata (quando vengono raggiunti i partecipanti richiesti).

La reservation è un **Observable** del design pattern *Observer*. Infatti il ciclo di notifica dell'observer tramite il metodo **notifyObserver()** viene chiamato all'interno del metodo **setConfirmed(...)**, come mostrato nello Snippet 2.

```
1 public void setConfirmed(boolean confirmed) throws SQLException {  
2     isConfirmed = confirmed;  
3     notifyObserver();  
4 }
```

Snippet 2: Implementazione del metodo **setConfirmed(...)** di **Reservation** nel DomainModel.

3.1.8 Person

Questa classe astratta rappresenta una persona racchiudendo i propri dati anagrafici, quindi i campi e metodi saranno usate dalle classi figlie.

I campi sono:

- **Id**: l'id nel database;
- **Email**: l'email dell'account.
- **Username**: l'username dell'account
- **Password**: la password dell'account.
- **City**: la città di residenza associata all'account.
- **Province**: la provincia di residenza associata all'account.
- **Zip**: il CAP di residenza associato all'account.
- **Country**: la nazione di residenza associata all'account.
- **Target**: il tipo di account (*User* o *Owner*). Utile per identificare in modo rapido la tipologia di account.

3.1.9 User

Questa classe estende la classe **Person**, e rappresenta l'entità **User**. Nonostante usi lo stesso costruttore di **Person**, passandogli "User" come **target**, è concettualmente e logicamente più corretto differenziare le due classi soprattutto in caso di aggiunta di comportamenti differenti.

3.1.10 Owner

Questa classe estende la classe **Person**, e rappresenta l'entità **Owner**. Nonostante usi lo stesso costruttore di **Person**, passandogli "Owner" come **target**, è concettualmente e logicamente più corretto differenziare le due classi soprattutto in caso di aggiunta di comportamenti differenti.

3.1.11 WorkingHours

Questa classe rappresenta l'entità **WorkingHours**. Un'oggetto di questa classe rappresenta uno slot orario contiguo appartenente ad una determinata Facility. Quindi se una Facility non fa orario continuato, ad esempio apre dalle 8:00 alle 13:00 e poi riapre alle 15:00 fino alle 20:00. Avrà due oggetti differenti rappresentanti due slot orari contigui, ovvero senza chiusura prima dell'orario di chiusura dello slot. I campi della classe sono:

- **Id**: id nel database;
- **DayOfWeek**: Il giorno della settimana di uno slot orario contiguo
- **OpeningHours**: l'orario di apertura di uno slot orario contiguo.
- **ClosingHours**: l'orario di chiusura di uno slot orario contiguo.

3.1.12 Sport

Questa classe rappresenta un'entità **sport**, il quale può essere assegnato in modo non univoco ad uno o più campi. I campi della classe sono:

- **Id**: l'id nel database;
- **Name**: il nome dello sport.
- **PlayersRequired**: il numero di giocatori richiesti.

3.1.13 Invite

La classe rappresenta l'entità **Invite**, ovvero un invito ricevuto dall'utente per unirsi ad un gruppo. Essa eredita da **Product** ed è quindi uno dei prodotti derivati. Gli attributi della classe sono :

- **Id**: l'id nel database;
- **Group**: il gruppo a cui fa riferimento l'invito.
- **User**: l'utente a cui è destinato.

3.1.14 Product

Questa è la classe astratta dal quale erediteranno tutte le classi che devono essere instanziate usando il pattern factory ovvero la classe **Creator** e le sue derivate. Essa rappresenta un prodotto generico e presenta solo il metodo astratto **build()**.

3.1.15 Creator

Creator è la classe astratta che definisce una *factory*, essa presenta il metodo astratto **createProduct()**, il quale verrà implementato dalle classi derivate e il **factoryMethod()** che instanzia l'oggetto, lo inizializza tramite **build()** e lo restituisce al chiamante.

3.1.16 InviteSender

Questa classe estende la classe **Creator** e oltre ad ampliare il **factoryMethod()** implementa **createProduct()** andando a restituire il **Product** derivato (Snippet 3). Si occupa anche di inizializzare gli attributi del singolo invito. La classe presenta l'attributo **groupSender** ovvero il gruppo di riferimento dell'invito il quale viene settato alla costruzione della classe.

```

1 public class InviteSender extends Creator {
2
3     private final Group groupSender;
4
5     public InviteSender(Group groupSender) {
6         this.groupSender = groupSender;
7     }
8
9     // methods
10    @Override
11    public Invite factoryMethod() {
12        Product product = super.factoryMethod();
13        Invite invite;
14        if (product instanceof Invite) {
15            invite = (Invite) product;
16            invite.setGroup(groupSender);
17            return invite;
18        }
19        return null;
20    }
21
22    @Override
23    public Invite createProduct() {
24        return new Invite();
25    }
26
27 }

```

Snippet 3: Implementazione della classe **InviteSender** della Domain Model.

3.1.17 NotificationSender

Questa classe estende la classe **Creator** e oltre ad ampliare il **factoryMethod()** implementa **createProduct()** andando a restituire il **Product** derivato. Si occupa anche di inizializzare gli attributi della singola notifica. La classe presenta gli attributi:

- **ReservationSender**: la prenotazione a cui fa riferimento;
- **NotificationType**: il tipo di notifica.
- **NotificationMessage**: il contenuto della stessa.

I vari attributi vengono settati alla costruzione della classe.

3.1.18 Field

Rappresenta l'entità **Field** ovvero un campo appartenente ad una **Facility**, esso potrà essere prenotato dai vari attori. Gli attributi della classe sono:

- **Id**: l'id nel database;
- **Name**: il nome del campo.
- **Image**: la stringa contenente i nome dell'immagine.
- **Facility**: lo stabilimento di appartenenza.

3.1.19 Facility

La classe si riferisce all'entità **Facility** ovvero ad un impianto sportivo, il quale apparterrà ad un **Owner** e comprenderà uno o più campi. Inoltre sarà caratterizzato da degli orari di apertura Essa presenta gli attributi:

- **Id**: l'id nel database;
- **Name**: il nome dello stabilimento.
- **Address**: l'indirizzo.
- **City**: la città.

- **Province**: la provincia.
- **Zip**: il codice zip.
- **Country**: la nazione.
- **NManagers**: il numero di manager registrati.
- **Fields**: la lista dei campi.
- **Telephone**: il numero di telefono.
- **Image**: la stringa contenente i nome dell'immagine.
- **WorkingHours**: le ore di apertura.
- **Owner**: il proprietario.

3.2 Business Logic

In questo package sono implementate le classi che gestiscono la logica di business del sistema.

Il package **BusinessLogic**, come mostrato in figura 1, è l'unico ad interfacciarsi con i DAO, in modo tale da non rompere l'isolamento e la responsabilità dei vari packages. Ogni classe di questo package che implementa le funzionalità collegate ai vari attori prende la sessione dal **SessionController**, il quale si occupa di mantenerla durante tutta la vita dell'applicazione. La sessione viene presa dalle varie classi al momento della loro creazione.

3.2.1 PasswordEncoder

Questa classe, composta solo da **metodi statici**, serve per codificare password con l'algoritmo *SHA-256*, basato su hash non reversibili. È stato scelto volutamente questo algoritmo per semplicità e per evitare di salvare password in chiaro nei database. Sono presenti solo due metodi:

- **HashPassword(...)**: codifica una password con *SHA-256* e restituisce l'hash.
- **VerifyPassword(...)**: confronta una password non codificata con un hash e restituisce **true** o **false**.

Quindi una volta codificata, una password non può essere decodificata, ma solo confrontata.

3.2.2 ProfileController

Questa classe astratta rappresenta la logica di business della sezione Profile per la gestione del proprio account. L'utilizzo dei **Java Generics** permettono di unificare i campi e metodi comuni con le classi figlie.

La classe ha due campi: **Person** (tipo T) e **PersonDAO** (tipo D), inizializzati dalle classi figlie di **ProfileController**.

I metodi getter, come **getEmail()**, recuperano i dati dal campo **Person**. I metodi di update, come **updateEmail(...)**, modificano i campi nel DomainModel e nel database, come mostrato nello Snippet 4.

```

1  public boolean updateEmail(String newEmail) throws SQLException {
2      try {
3          personDao.updateEmail(person.getUsername(), newEmail);
4          this.person.setEmail(newEmail);
5          System.out.println("Email updated");
6      } catch (SQLException e) {
7          e.printStackTrace();
8          return false;
9      }
10     return true;
11 }
```

Snippet 4: Implementazione del metodo **updateEmail(...)** di **ProfileController** nella **BusinessLogic**.

Infine, è presente il metodo **logOut()** che permette di effettuare il logout e tornare alla schermata di accesso, come mostrato nello Snippet 5.

```

1 public void logOut() {
2     SessionController.getInstance().setPerson(null);
3     this.person = null;
4 }

```

Snippet 5: Implementazione del metodo **logOut(...)** di **ProfileController** nella BusinessLogic.

3.2.3 UserProfileController

Questa classe estende la classe astratta **PersonController**, che utilizza i **Java Generics**, come mostrato nello Snippet 6. Le vengono passati **User** (per **Person**) e **UserDAO** (per **PersonDAO**). Presenta un solo campo, **ManagesDAO**, che viene utilizzato dal metodo **getFacilitiesManaged()** per ottenere gli impianti sportivi.

```

1 public class UserProfileController extends ProfileController<User, UserDAO> {
2 // ...
3 // ...

```

Snippet 6: Dichiarazione della classe **UserProfileController** della BusinessLogic.

3.2.4 OwnerProfileController

Questa classe estende **PersonController** passando i propri valori per i **Java Generics**, come mostrato nello Snippet 7. Le vengono passati **Owner** (per **Person**) e **OwnerDAO** (per **PersonDAO**).

```

1 public class OwnerProfileController extends ProfileController<Owner, OwnerDAO> {
2 // ...
3 // ...

```

Snippet 7: Dichiarazione della classe **OwnerProfileController** della BusinessLogic.

3.2.5 PersonController

Questa classe astratta definisce le azioni eseguibili da ogni tipo di attore. Utilizza i **Java Generics** sul campo **Person**, come in **ProfileController**. I campi includono DAO e **NotificationController** per la gestione delle notifiche.

I metodi di get, come **getGroupByReservation(...)**, si interfacciano con i DAO e sono utili a tutti gli utenti.

I metodi **addGroupMemeber(...)**, **changeUserGuests(...)** e **removeGroupMember(...)** gestiscono l'aggiunta, la modifica e la rimozione di membri del gruppo (compresi i guests).

I metodi **sendInvite(...)** e **sendInvites(...)** inviano un singolo invito o più inviti, rispettivamente. Il secondo utilizza il primo in un loop.

Questi metodi vengono chiamati da **addReservation(...)** (Snippet 8), che riceve i parametri della **Reservation** in input e restituisce l'ID della prenotazione creata con successo o 0 in caso di errore.

Nonostante le differenze di logica in base al tipo di account, il polimorfismo di Java consente una singola implementazione delegando le differenze ai metodi astratti **joinGroupHelper(...)**, **applyChangesFromDraft(...)** e **getProvinceForMatching(...)**, lasciando l'implementazione alle classi figlie dato che si devono comportare diversamente a seconda del tipo di account. I metodi **checkReservationData(...)** e **checkGroupData(...)** verificano i dati di prenotazione e gruppo. In caso di fallimento, la prenotazione fallisce.

Il metodo usa una transaction sul database per mantenere l'integrità dei dati. Durante la transaction, le modifiche al **DomainModel** sono in bozza e vengono applicate dopo il commit tramite **applyChangesFromDraft(...)** di **Group** dato che in questo caso è l'unico oggetto a subire modifiche. Questo può essere fatto grazie ad una deep copy degli oggetti interessati. La gestione delle transactions viene trattata più in dettaglio nel paragrafo 3.5.3.

Il metodo **joinGroupHelper(...)** è astratto e necessita di override. La classe **UserActionsController** (lato utente) chiama il suo metodo **joinGroup(...)**. Invece, la classe **ManagerOwnerManagementController** (lato manager e owner) restituisce sempre *true*, bypassando la verifica di **joinGroup(...)** non presente per manager e owner, che agiscono come terzi.

Il metodo astratto **getProvinceForMatching(...)** assegna la corretta provincia in base al tipo di account che aggiunge una prenotazione. Questo input viene passato al metodo **findOtherPlayers(...)** per cercare i giocatori a cui inviare inviti in base alla provincia.

```

1   public int addReservation(Date eventDate, Time eventTimeStart, Time
2     eventTimeEnd, Field field, int guests, int requiredParticipants, boolean
3     isMatched, User groupHead, ArrayList<GroupMember> removedMember, ArrayList<
4     GroupMember> addedMember, ArrayList<GroupMember> changedMember, ArrayList<
5     String> inviteList) {
6     Reservation reservation = new Reservation(eventDate, eventTimeStart,
7     eventTimeEnd, field, isMatched);
8
9     if (checkReservationData(reservation)) {
10       try {
11         //start transaction
12         reservationDao.getConnection().setAutoCommit(false);
13
14         //execute queries
15         int newReservationId = reservationDao.addReservation(reservation
16       );
17         reservation.setId(newReservationId); //WARNING: it's very
18         important
19
20         //group creation
21         Group group = new Group(groupHead, reservation,
22         requiredParticipants, guests);
23         if (checkGroupData(group)) {
24
25           int newGroupId = groupDao.addGroup(group);
26           group.setId(newGroupId); //WARNING: it's very important
27
28           Group draftGroup = SerializationUtils.clone(group); //deep
29           copy for DM transaction
30
31           if (joinGroupHelper(draftGroup, guests)) {
32
33             if (isMatched) {
34               int invitesSent = sendInvites(draftGroup,
35               findOtherPlayers(getProvinceForMatching(field)));
36               if (invitesSent >= 0)
37                 System.out.println("Invites sent: " +
38               invitesSent);
39               else
40                 System.out.println("Error while sending invites"
41             );
42
43             } else {
44               notificationController.sendConfirmNotifications(
45               reservation);
46             }
47           } else {
48             throw new TransactionException("Error while joining into
49             group");
50           }
51
52           try {
53             if (!applyChangesFromDraft(draftGroup, removedMember,
54             addedMember, changedMember, guests, inviteList, null)) {
55               throw new TransactionException("Error while applying
56               changes");
57             }
58           } catch (SQLException | ClassNotFoundException e3) {
59             throw new TransactionException("Error while applying
60               changes");
61           }
62
63           //commit transaction
64           reservationDao.getConnection().commit();
65
66           //apply group changes
67           group.applyChangesFromDraft(draftGroup);
68
69           return newReservationId;
70         }
71       } else {
72         throw new TransactionException("Wrong group data");
73       }
74     }
75   }

```

```

60         } catch (SQLException | ClassNotFoundException | 
61 TransactionException e) {
62     try {
63         //general rollback
64         reservationDao.getConnection().rollback();
65 
66         return 0;
67     } catch (SQLException e1) {
68         e1.printStackTrace();
69     }
70     } finally {
71 
72     try {
73         //end transaction
74         reservationDao.getConnection().setAutoCommit(true);
75     } catch (SQLException e1) {
76         e1.printStackTrace();
77     }
78 }
79 
80     return 0;
81 }
82 }
```

Snippet 8: Implementazione del metodo **addReservation(...)** della classe **PersonController** della BusinessLogic.

Un altro metodo rilevante è **editReservation(...)**, che differisce per la chiamata al metodo **applyChangesFromDraft(...)**, con implementazione diversa a seconda del tipo di account. Anche **editReservation(...)** utilizza una transaction per mantenere l'integrità dei dati, come mostrato nello Snippet 9. Anche in questo caso le modifiche al **DomainModel** avvengono in bozza durante la transaction.

```

1  public boolean editReservation(Group group, ArrayList<GroupMember>
2      removedDraft, ArrayList<GroupMember> addedDraft, ArrayList<GroupMember>
3      changedDraft, int ownGuests, ArrayList<String> inviteList, String
4      newGroupHeadUsername) {
5 
6     Reservation previousReservation = null;
7 
8     try{
9         previousReservation = reservationDao.getReservation(group.
10        getReservation().getId(), false);
11    } catch (SQLException | ClassNotFoundException e) {
12        return false;
13    }
14 
15    try {
16        //start transaction
17        isPartDao.getConnection().setAutoCommit(false);
18 
19        reservationDao.updateEventDate(group.getReservation().getId(), group.
20        getReservation().getEventDate());
21        reservationDao.updateEventTimeEnd(group.getReservation().getId(),
22        group.getReservation().getEventTimeEnd());
23        reservationDao.updateEventTimeStart(group.getReservation().getId(),
24        group.getReservation().getEventTimeStart());
25 
26        Group draftGroup = null;
27 
28        boolean changesHasBeenApplied = false;
29        try {
30            draftGroup = SerializationUtils.clone(group); //deep copy for DM
31            transaction
32 
33            changesHasBeenApplied = applyChangesFromDraft(draftGroup,
34            removedDraft, addedDraft, changedDraft, ownGuests, inviteList,
35            getUserByUsername(newGroupHeadUsername));
36        } catch (SQLException | ClassNotFoundException e2) {
37            throw new TransactionException(e2.getMessage());
38        }
39 
40        if (!changesHasBeenApplied)
41            throw new TransactionException("Error while applying changes");
42    }
```

```
33
34     //commit transaction
35     isPartDao.getConnection().commit();
36
37     System.out.println("Participant OLD: " + group.getParticipants());
38     //apply group changes
39     if (draftGroup != null)
40         group.applyChangesFromDraft(draftGroup);
41
42     System.out.println("Participant NEW: " + group.getParticipants());
43
44
45     notificationController.sendModificationNotifications(group.
getReservation());
46     return true;
47 } catch (SQLException | TransactionException e) {
48
49     try {
50         //rollback transaction
51         isPartDao.getConnection().rollback();
52     } catch (SQLException e1) {
53         e1.printStackTrace();
54     }
55
56     return false;
57 } finally {
58     try {
59         //end transaction
60         isPartDao.getConnection().setAutoCommit(true);
61     } catch (SQLException e1) {
62         e1.printStackTrace();
63     }
64 }
65
66
67 }
```

Snippet 9: Implementazione del metodo `editReservation(...)` della classe `PersonController` della BusinessLogic.

3.2.6 UserActionsController

Questa classe estende **PersonController** e rappresenta le azioni che posso compiere gli utenti.

Il metodo `acceptInvite(...)` (Snippet 10) permette agli utenti di accettare inviti. Per le prenotazioni **“matched”**, chiede se inviare inviti manualmente e unirsi al gruppo. Questo non è possibile per le prenotazioni **“not matched”**. Una transaction mantiene l'integrità dei dati.

```

26             if (!joinGroupHelper(invite.getGroup(), 0))
27                 return false;
28
29         }
30
31
32         //delete this invite
33         inviteDao.deleteInvite(invite.getId());
34
35         //commit transaction
36         userDao.getConnection().commit();
37
38         accepted = true;
39     } catch (SQLException | ClassNotFoundException e) {
40         try{
41             //rollback transaction
42             userDao.getConnection().rollback();
43         } catch (SQLException e1){
44             e1.printStackTrace();
45         }
46     } finally {
47         try {
48             userDao.getConnection().setAutoCommit(true);
49         } catch (SQLException e) {
50             e.printStackTrace();
51         }
52     }
53
54     return accepted;
55 }
```

Snippet 10: Implementazione del metodo **acceptInvite(...)** della classe **UserActionsController** della BusinessLogic.

Il metodo **leaveGroup(...)** permette agli utenti di uscire da un gruppo (portando con sé i propri guests senza account). Una transaction mantiene l'integrità dei dati. Se il capo gruppo lascia, viene assegnato un successore. Se il gruppo è vuoto, viene eliminato (con la relativa prenotazione). Nello Snippet 11 viene mostrata l'implementazione del metodo.

```

1  public boolean leaveGroup(int idGroup) {
2
3      Group group = null;
4
5      try {
6          group = groupDao.getGroup(idGroup);
7      }
8      catch (SQLException | ClassNotFoundException e) {
9          return false;
10     }
11
12     //this method removes a member from DomainModel
13     boolean memberRemoved = group.removeMember(person);
14
15     if (memberRemoved) {
16         try {
17             //start transaction
18             isPartDao.getConnection().setAutoCommit(false);
19
20             //execute queries
21
22             isPartDao.removeMembership(idGroup, person.getId());
23
24             if (group.getParticipants() <= 0) {
25                 //set reservation as deleted (and related group). It will be
26                 deleted by trigger.
27                 boolean deletedSuccessfully = deleteReservation(group.
28                 getReservation().getId());
29
30                 if (!deletedSuccessfully) {
31                     throw new TransactionException("Error while deleting.");
32                 }
33                 else
34                     groupDao.updateGroupHead(idGroup, group.getGroupHead().getId
35
36             ());
37         }
38     }
39 }
```

```

35         //commit transaction
36         isPartDao.getConnection().commit();
37     }
38     catch (SQLException | TransactionException e) {
39
40         try {
41             //rollback transaction
42             isPartDao.getConnection().rollback();
43
44             } catch (SQLException e1) {
45                 e1.printStackTrace();
46             } finally {
47                 try {
48                     //end transaction
49                     isPartDao.getConnection().setAutoCommit(true);
50                 } catch (SQLException e1) {
51                     e1.printStackTrace();
52                 }
53             }
54
55             return false;
56         }
57     }
58     else {
59         System.out.println("Error during removing");
60         return false;
61     }
62
63     return true;
64 }
```

Snippet 11: Implementazione del metodo **leaveGroup(...)** della classe **UserActionsController** della BusinessLogic.

Il metodo **editRights(...)** (Snippet 12) stabilisce chi può modificare una prenotazione. Questa verifica avviene prima di mostrare i tasti di modifica.

```

1  public boolean editRights(Reservation reservation) throws SQLException,
2      ClassNotFoundException {
3      boolean pass = true;
4      Group group = groupDao.getGroupByReservation(reservation.getId());
5
6      if(!group.getGroupHead().getUsername().equals(person.getUsername())) {
7          pass = false;
8      }
9      if(reservation.isMatched()) {
10         pass = false;
11     }
12
13     if(reservation.getEventTimeStart().toLocalTime().getHour() - LocalTime.
14     now().getHour() < 2 && reservation.getEventDate().toLocalDate().equals(
15     LocalDate.now())){
16         pass = false;
17     }
18
19     return pass;
20 }
```

Snippet 12: Implementazione del metodo **editRights(...)** della classe **UserActionsController** della BusinessLogic.

I metodi **addReservation(...)** e **editReservation(...)** chiamano **applyChangesFromDraft(...)** per “fare un push” dalle bozze al database, come mostrato nello Snippet 13. Questo gestisce l’aggiunta, la modifica, la rimozione di membri dal gruppo e l’invio di inviti in base ai permessi degli utenti (Figura 2).

```

1  @Override
2  public boolean applyChangesFromDraft(Group group, ArrayList<GroupMember>
3      removedDraft, ArrayList<GroupMember> addedDraft, ArrayList<GroupMember>
4      changedDraft, int ownGuestsSelected, ArrayList<String> inviteListDraft, User
5      newGroupHead) throws SQLException, ClassNotFoundException {
6
7      if (group != null) {
8
9          if (inviteListDraft != null && !inviteListDraft.isEmpty()) {
```

```
int invitesSent = sendInvites(group, getUsersByUsernames(
inviteListDraft));

if (invitesSent < 0)
    return false;
}

//observer attach
group.getReservation().attach(notificationController);

boolean areGuestsChanged = changeOwnGuests(group, ownGuestsSelected);
;

if (!areGuestsChanged)
    return false;

boolean success = true;

if (removedDraft != null && !removedDraft.isEmpty() && group.
getGroupHead().getUsername().equals(person.getUsername()))
    success = removeGroupMembers(group, removedDraft);

return success;
}

return true;
}
```

Snippet 13: Implementazione del metodo `applyChangesFromDraft(...)` della classe `UserActionsController` della BusinessLogic.

3.2.7 ManagerOwnerManagementController

Questa classe eredita dalla classe **PersonController** e implementa le funzioni utilizzabili sia dai proprietari che dai manager. Permette di effettuare annunci ai partecipanti di un gruppo tramite **reservationAnnouncement(...)** e fa l'override del metodo **joinGroupHelper(...)** della classe base. Presenta anche altri metodi come per esempio **getWHsByFacilityByDay(...)**, il quale restituisce gli orari di apertura di una **Facility** dato un giorno e **getFieldsByFacility(...)** che invece restituisce le prenotazione dato un campo.

3.2.8 OwnerManagementController

La classe eredita dalla classe **ManagerOwnerManagementController** e gestisce le funzioni che necessitano esclusivamente i proprietari. Presenta vari metodi per l'ottenimento di dati statistici come **monthlyReservations()** e **dailyEarnings()**. Poi sono presenti per la gestione dei manager data una **Facility** come **getManagersByFacility(...)** e **attachManager(...)**

Infine implementa le funzioni per la creazione, modifica e cancellazione di **Field**, **Sport**, **Facility** e **WorkingHours** per esempio **addFacility(...)**, **editFacility(...)**. La classe presenta gli attributi **managesDAO** e **sportDao** per la connessione al database e **weekDays** per le statistiche;

3.2.9 NotificationController

Questa classe implementa l'interfaccia **Observer**. I suoi campi sono i DAO, **Reservation** e **Person**.

Implementando l'**Observer**, bisogna anche implementare il metodo **update(...)**, come mostrato nello Snippet 14. Questo metodo notifica i membri del gruppo, i gestori e proprietari dell'impianto sportivo, e imposta le flag **IsConfirmed** e **IsNotified** a **true**, indicando rispettivamente la conferma della prenotazione e l'invio delle notifiche.

```
1  @Override  
2  public void update(Observable observable) throws SQLException {  
3  
4      if (observable instanceof Reservation){  
5          Reservation reservation = (Reservation) observable;  
6  
7          if (reservation.isConfirmed() && reservation.isMatched() && !  
reservation.isNotified()) {  
8  
9              reservationDao.updateIsConfirmed(reservation.getId(), true);  
10         }  
11     }  
12 }
```

```

10             reservationDao.updateIsNotified(reservation.getId(), true);
11             sendConfirmNotifications(reservation);
12             reservation.considerNotified();
13         }
14     }
15 }
16
17 }
18 }
```

Snippet 14: Implementazione del metodo **update(...)** della classe **NotificationController** della BusinessLogic.

Il metodo protetto **sendNotifications(...)** invia le notifiche e restituisce il numero di notifiche inviate (numero intero maggiore o uguale a **0**) o **-1** in caso di errore, come mostrato nello Snippet 15.

```

1 protected int sendNotifications(Reservation reservation, NotificationType
2 notificationType, String notificationMessage) {
3
4     Owner owner;
5     Facility facility;
6
7     int count = 0;
8
9     if(notificationType != NotificationType.ANNOUNCEMENT) {
10         notificationMessage = null;
11     }
12
13     NotificationSender notificationSender = new NotificationSender(
14         reservation, notificationType, notificationMessage);
15     Notification tmpNotification;
16
17     try {
18         facility = facilityDAO.getFacility(reservation.getField().
19             getFacility().getId(), false);
20         owner = ownerDAO.getOwnerByID(facility.getOwner().getId());
21     } catch (SQLException e) {
22         return -1;
23     }
24
25     try {
26         tmpNotification = notificationSender.factoryMethod();
27         tmpNotification.setRecipient(owner);
28         notificationDAO.addNotification(tmpNotification);
29         count++;
30     } catch (SQLException e) {
31
32     }
33
34     ArrayList<User> managers = new ArrayList<>();
35     ArrayList<User> invitablesUsers = new ArrayList<>();
36     try {
37         managers = managesDAO.getAllManagersByFacility(facility.getId());
38         invitablesUsers.addAll(Group.getUsersByGroupMembers(isPartDao.
39             getGroupMembers(groupDAO.getGroupByReservation(reservation.getId()).getId())));
40     }
41     invitablesUsers.removeAll(managers);
42 } catch (SQLException | ClassNotFoundException e) {
43
44 }
45
46 for(User user : managers){
47     try {
48         tmpNotification = notificationSender.factoryMethod();
49         tmpNotification.setRecipient(user);
50         notificationDAO.addNotification(tmpNotification);
51         count++;
52     } catch (SQLException e) {
53
54     }
55 }
56
57 for (User user:invitablesUsers){
58     try {
59         tmpNotification = notificationSender.factoryMethod();
60
61     }
```

```

56             tmpNotification.setRecipient(user);
57             notificationDAO.addNotification(tmpNotification);
58             count++;
59         }
60     } catch (SQLException e) {
61
62     }
63 }
64
65     if (count == 0 && owner != null && !managers.isEmpty() && !
66     invitablesUsers.isEmpty()){
67         return -1;
68     }
69
70     return count;
71 }
```

Snippet 15: Implementazione del metodo **sendNotifications(...)** della classe **NotificationController** della BusinessLogic.

Questo metodo non è pubblico perché usa metodi **helper** come **sendConfirmNotification(...)** (Snippet 16) per ridurre i parametri. Questo vale per ogni **NotificationType**.

```

1 public int sendConfirmNotifications(Reservation reservation) {
2     return sendNotifications(reservation, NotificationType.CONFIRMATION, "");
3 }
```

Snippet 16: Implementazione del metodo **sendConfirmNotifications(...)** della classe **NotificationController** della BusinessLogic.

La notifica Announcement richiede anche un messaggio personalizzato dall'helper **sendAnnouncement(...)**, come mostrato nello Snippet 17.

```

1 public int sendAnnouncements(Reservation reservation, String message) {
2     return sendNotifications(reservation, NotificationType.ANNOUNCEMENT,
3     message);
4 }
```

Snippet

17:

Implementazione del metodo **sendAnnouncements(...)** della classe **NotificationController** della BusinessLogic.

Il metodo **deleteNotification(...)** si occupa di eliminare una notifica dal database, chiamando il DAO e restituendo un booleano.

3.2.10 AccessStrategy

Questa classe è l'interfaccia che gestisce il login e la registrazione degli utenti. Lo fa tramite le funzioni di **login()**, **register()**. Le funzioni **checkEmail()**, **checkPassword()** e **checkPersonExistence()** svolgono il controllo dei dati inseriti e delle credenziali (Snippet 18).

```

1 public interface AccessStrategy {
2
3     //methods
4     Person login(String username, String notEncodedPassword) throws SQLException,
5     , NoSuchAlgorithmException, ClassNotFoundException;
5     boolean checkPersonExistence(String username) throws SQLException,
6     ClassNotFoundException ;
6     boolean checkEmail(String emailEntered) throws SQLException,
7     ClassNotFoundException;
7     boolean register(String username, String email, String password, String city
8     , String province, String zip, String country);
8     boolean checkPassword(String username, String notEncodedPassword) throws
9     SQLException, ClassNotFoundException, NoSuchAlgorithmException;
9 }
```

Snippet 18: Implementazione della classe **AccessStrategy** della Business Logic.

3.2.11 UserAccess

Questa classe implementa l'interfaccia **AccessStrategy** gestendo il login e la registrazione degli utenti attraverso le funzioni di tale interfaccia; Essa usa la classe **UserDAO** per comunicare con il database.

3.2.12 OwnerAccess

La classe implementa **AccessStrategy** e permette il login e la registrazione dei proprietari attraverso l'interfaccia. Essa usa la classe **OwnerDAO** per comunicare con il database.

3.2.13 AccessController

Questa classe permette di implementare il pattern *Strategy* e quindi di decidere al momento della creazione della stessa quale implementazione usare, essa si limita solo a chiamare i metodi dell'interfaccia **AccessStrategy** di una delle due implementazioni (Snippet 19).

```
1 public class AccessController {
2     // attributes
3     private AccessStrategy accessStrategy;
4
5     //constructor
6
7     public AccessController(AccessStrategy accessStrategy) {
8         this.accessStrategy = accessStrategy;
9     }
10
11    // methods
12
13    public AccessStrategy getAccessStrategy() {
14        return accessStrategy;
15    }
16
17    public void setAccessStrategy(AccessStrategy accessStrategy) {
18        this.accessStrategy = accessStrategy;
19    }
20
21    public Person login(String username, String notEncodedPassword) throws
22        SQLException, NoSuchAlgorithmException, ClassNotFoundException {
23        return accessStrategy.login(username, notEncodedPassword);
24    }
25
26    public boolean checkPassword(String username, String password) throws
27        SQLException, NoSuchAlgorithmException, ClassNotFoundException {
28        return accessStrategy.checkPassword(username, password);
29    }
30
31    public boolean checkPersonExistence(String username) throws SQLException,
32        ClassNotFoundException {
33        return accessStrategy.checkPersonExistence(username);
34    }
35
36    public boolean checkEmail(String emailEntered) throws SQLException,
37        ClassNotFoundException {
38        return accessStrategy.checkEmail(emailEntered);
39    }
40
41    public boolean register(String username, String email, String password,
42        String city, String province, String zip, String country){
43        return accessStrategy.register(username, email, password, city, province
44        , zip, country);
45    }
46}
```

Snippet 19: Implementazione della classe **AccessController** della Business Logic.

3.2.14 SessionController

La classe implementa il design pattern *Singleton* e si occupa di salvare, attraverso il metodo **setPerson()**, e mantenere la sessione dell'utente una volta effettuato il login. Attraverso i metodi **getPerson()** e **getInstance()** fornisce la sessione e l'utente attualmente connesso alle classi della **BusinessLogic** che ne fanno richiesta (Snippet 20).

```
1 public class SessionController {
2
3     private Person person = null;
4
5     private static final SessionController instance = new SessionController();
6
7     private SessionController() {}
8
9     public static SessionController getInstance() {
```

```

10     return instance;
11 }
12
13 public Person getPerson() {
14     return person;
15 }
16
17 public void setPerson(Person person) {
18     this.person = person;
19 }
20 }
```

Snippet 20: Implementazione della classe **SessionController** della Business Logic.

3.3 Object-Relational Mapping (ORM)

3.3.1 ConnectionManager

Questa classe (mostrata nello Snippet 21, implementa il design pattern *Singleton* e funge da gestore per la connessione al database **PostgreSQL**. Infatti, come mostrato in questo snippet, i campi di classe sono:

- **Url**: l'indirizzo del database **PosgresSQL**.
- **Username**: username per accedere al database.
- **Password**: password per accedere al database.
- **Connection**: campo che tiene la connessione con il database, inizialmente nullo.
- **Instance**: istanza unica del pattern singleton.

Tutti i questi campi sono **statici**.

Invece i metodi sono:

- **getInstance()**: metodo statico del pattern singleton che restituisce l'istanza, se non esiste la crea e la mantiene.
- **getConnection()**: metodo che restituisce la connessione stabilita con il database.

```

1 public class ConnectionManager {
2     private static final String url = "jdbc:postgresql://localhost:5432/
3         elaboratoswe_db";
4     private static final String username = "postgres";
5     private static final String password = "postgres";
6     private static Connection connection = null;
7
8     // singleton instance
9     private static ConnectionManager instance = null;
10
11    private ConnectionManager(){}
12
13    public static ConnectionManager getInstance() {
14
15        if (instance == null) { instance = new ConnectionManager(); }
16
17        return instance;
18    }
19
20    public Connection getConnection() throws SQLException,
21    ClassNotFoundException {
22
23        Class.forName("org.postgresql.Driver");
24
25        if (connection == null)
26            try {
27                connection = DriverManager.getConnection(url, username, password
28            );
29                //connection.setAutoCommit(false);
30            } catch (SQLException e) {
31                System.err.println("Error: " + e.getMessage());
32            }
33    }
34 }
```

```

31         return connection;
32     }
33 }

```

Snippet 21: Implementazione della classe **ConnectionManager** dell'ORM.

3.3.2 ConnectionHolder

Questa **classe astratta**, mostrata nello Snippet 22, serve a gestire tutte le connessioni di tutti i DAO. Infatti ogni DAO estende **ConnectionHolder**. Questa classe si occupa principalmente di instaurare la connessione con il database usando il *singleton* **ConnectionManager** durante la costruzione del DAO figlio. Poi è possibile anche ottenere la connessione instaurata se vogliamo gestirla manualmente dai chiamanti.

Ha un campo protetto **Connection**, ovvero la connessione. Il **costruttore** è una parte fondamentale perché serve alle classi figlie per instaurare la connessione (con il DB tramite il ConnectionManager) e non avere ridondanze di codice.

Invece il metodo getter **getConnection()** permette di ottenere una connessione dai chiamanti, come mostrato al paragrafo 2.6.3.

```

1 public abstract class ConnectionHolder {
2
3     protected Connection connection;
4
5     //constructor
6
7     public ConnectionHolder() {
8         try {
9             this.connection = ConnectionManager.getInstance().getConnection();
10        } catch (SQLException | ClassNotFoundException e) {
11            System.err.println("Error: " + e.getMessage());
12        }
13    }
14
15    //getters
16    public Connection getConnection() {
17        return connection;
18    }
19 }

```

Snippet 22: Implementazione della classe **ConnectionHolder** della BusinessLogic.

3.3.3 FacilityDAO

Questa classe si occupa della gestione dei dati della Facility, eseguendo azioni sul database. Aggiunge e rimuove istanze con i metodi **addFacility(...)** e **deleteFacility(...)**. Invece con i metodi **getFacility(...)** e **getFacilitiesByOwner(...)** vengono estratti gli impianti richiesti. Invece per modificare gli attributi di una Facility, è possibile utilizzare un metodo di update per ogni attributo da modificare, come **updateCountry(...)** o **updateAddress(...)**.

3.3.4 ManagesDAO

Questa classe si occupa di associare o dissociare i Manager e una determinata Facility, eseguendo azioni sul database. Questo è possibile farlo rispettivamente con i metodi **attachManager(...)** e **detachManager(...)**. Invece con i metodi di getter **getAllFacilitiesByManager(...)** e **getAllManagersByFacility(...)** è possibile ottenere dati riguardo l'associazione Manager-Facility.

3.3.5 NotificationDAO

Questa classe si occupa di aggiungere, prelevare o eliminare le notifiche dal database, interfacciandosi con le tabelle **NotificationUser** e **NotificationOwner** del DB. Questo viene fatto con metodi come **addNotification(...)**, **getNotification(...)** o **deleteNotification(...)**. Questa classe utilizza anche dei metodi privati come helper al suo interno per semplificare la logica.

3.3.6 PersonDAO

Questa **classe astratta** si occupa della gestione dei dati di una persona, eseguendo azioni sul database. **PersonDAO** ha solamente un campo protetto, ovvero **target**. Ha la solita funzione di quello presente nella classe **Person** del DomainModel, ovvero quella di identificare la tipologia di persona in modo rapido per gestire efficientemente e correttamente le tabelle da interrogare.

In questa classe sono inoltre presenti tutti i metodi che servono per azioni comuni, indipendentemente dal tipo di persona. Un esempio è **deletePerson(...)** che serve per eliminare una determinata istanza da una tabella (identificata dal target). Ma non è presente nessun metodo di aggiunta perché questa azione viene delegata alle classi figlie. Sono presenti anche metodi getter come **getUser(...)** o **getUsersByProvince(...)** dato che ogni tipologia (User, Manager o Owner) di persona potrebbe richiedere un utente.

Invece per modificare gli attributi di una persona, è possibile utilizzare un metodo di update per ogni attributo da modificare, come **updateUsername(...)** o **updateAddress(...)**.

Un metodo rilevante è **checkEmailExistence(...)** che serve per controllare se esistono persone con quella determinata email. Infatti viene utilizzato in fare si accesso per la verifica della email.

3.3.7 UserDAO

Questa classe estende PersonDAO, che si interfaccia con la tabella **User** del database. Praticamente rappresenta un utente. Infatti, a differenza di PersonDAO, questa classe ha il metodo **addUser(...)** che viene utilizzato in fase di registrazione per aggiungere un'istanza alla tabella del database. Tale metodo non è presente in PersonDAO perché viene utilizzato solamente in questo caso, quindi anche per motivi di sicurezza è utile averlo nella sua classe dedicata ovvero UserDAO. Ovviamente viene anche passato il target "*User*" al costruttore di PersonDAO.

3.3.8 OwnerDAO

Questa classe estende PersonDAO, che si interfaccia con la tabella **Owner** del database. Praticamente rappresenta un proprietario. Infatti, a differenza di PersonDAO, questa classe ha il metodo **addOwner(...)** che viene utilizzato in fase di registrazione per aggiungere un'istanza alla tabella del database. Tale metodo non è presente in PersonDAO perché viene utilizzato solamente in questo caso, quindi anche per motivi di sicurezza è utile averlo nella sua classe dedicata ovvero OwnerDAO, oltre al fatto che ha un portale di accesso separato. Ovviamente viene anche passato il target "*Owner*" al costruttore di PersonDAO. In più è interessante notare che il metodo getter **getOwner(...)** è presente solamente in questa classe, a differenza del metodo **getUser(...)** che è presente in PersonDAO. In questo modo possono utilizzarlo tutti. Questo perché l'ottenimento di un Owner da database viene fatto solamente quando siamo nel suo portale apposito, non quando accediamo come utenti.

3.3.9 WorkingHoursDAO

Questa classe gestisce i dati delle WorkingHours delle Facility, eseguendo azioni su database. Si interfaccia con la tabella **WH** del DB, gestendo gli slot orari per ogni Facility.

Nel modello SQL, la foreign key a Facility garantisce che ogni WorkingHours abbia una Facility associata. I vincoli di eliminazione e aggiornamento si estendono a cascata.

I vari tipi di metodi sono:

- Per aggiungere una istanza: **addWHToFacility(...)**.
- Per aggiornare un determinato slot orario: **updateWH(...)**.
- Per la rimozione: **removeWHFromFacility(...)**, **removeWHFromFacilityByDay(...)** e **removeAllWHsByFacility(...)**.
- Per il recupero dei dati: **getWH(...)**, **getWHsByFacility(...)** e **getWHsByFacilityByDay(...)**. Quest'ultimo è utile per ottenere gli slot orari di apertura di una Facility in un determinato giorno.

3.3.10 SportDAO

La classe si occupa di gestire i dati degli sport che si trovano nel database. Si occupa di recuperare i dati attraverso metodi quali **getSport(...)** e **getAllSport()**, mentre aggiunge o cancella sport con

addSport(...) e **deleteSport(...)**). Invece si occupa della modifica attraverso **updateSportPlayers(...)** e **updateSportName(...)**.

3.3.11 InviteDAO

Essa si occupa di gestire i dati nel database riguardanti gli inviti . Aggiunge e cancella inviti dal database attraverso **addInvite(...)** e **deleteInvite(...)**). Invece recupera i dati attraverso **getInvitesByUser(...)**. Infine si occupa di controllare se è già stato inviato un invito uguale ad un dato utente tramite **checkInvite(...)** (Snippet 23)..

```
1  public Boolean checkInvite(int idUser,int idGroup) throws SQLException,
2      ClassNotFoundException {
3
4      String querySQL = String.format("SELECT count(*) AS results FROM \""
5          "Invite\" WHERE id_user = '%d' AND id_group = '%d'",idUser, idGroup);
6
7      PreparedStatement preparedStatement = null;
8      ResultSet resultSet = null;
9
10     try {
11         preparedStatement = connection.prepareStatement(querySQL);
12         resultSet = preparedStatement.executeQuery();
13         if (resultSet.next()) {
14
15             int invites = resultSet.getInt("results");
16
17             if (invites > 0)
18                 return true;
19             } else{
20                 System.err.println("No invite found ");
21             }
22         } catch (SQLException e) {
23             System.err.println("Error: " + e.getMessage());
24         } finally {
25             if (preparedStatement != null) { preparedStatement.close(); }
26             if (resultSet != null) { resultSet.close(); }
27         }
28
29     return false;
30 }
```

Snippet 23: Implementazione della funzione **checkInvite()**.

3.3.12 IsPartDAO

Questa classe si occupa di gestire i dati riguardanti le partecipazioni degli utenti ai vari gruppi. Aggiunge e rimuove dati tramite **addMembership(...)** e **removeMembership(...)**). Invece recupera dati tramite metodi come **getGroupMembers(...)** e **getAllGroupsByUser(...)**. I dati possono essere modificati solo tramite **updateGuestsUsers(...)**.

3.3.13 FieldDAO

La classe gestisce i dati riguardanti i campi sportivi, aggiunge e rimuove dati tramite **addField(...)** e **deleteField(...)** mentre li recupera tramite metodi come **getField(...)** e **getFieldsByName(...)**. Per la modifica presenta vari metodi come per esempio **updateSport(...)** e **updateName(...)**.

3.3.14 GroupDAO

La classe gestisce i dati riguardanti i gruppi, aggiunge e rimuove dati tramite **addGroup(...)** e **deleteGroup(...)** mentre li recupera tramite **getGroup(...)** e **getGroupByReservation(...)**. L'unica modifica possibile è quella tramite il metodo **updateGroupHead(...)**.

3.3.15 ReservationDAO

Si occupa dei dati delle prenotazioni. Aggiunge e cancella prenotazioni tramite **addReservation(...)** e **deleteReservation(...)** mentre li recupera tramite metodi come **getReservationsByUser(...)** e **getReservation(...)**. Tramite metodi come **updateIsConfirmed(...)** e **updateEventDate(...)** è possibile modificare le prenotazioni. Essa fornisce anche dati statistici interessanti per il proprietario attraverso metodi come **dailyEarning(...)** e **dailyReservations(...)**.

3.4 GUI

I file FXML sono nel package **FXML**, mentre i GUIControl sono nel package **GUIControll**, all'interno di **FXML**.

Un singolo file FXML può essere associato solamente ad un **unico** controller GUI (**GUIControl**). Esistono classi astratte per ereditare proprietà comuni a più controller.

Tutte i controller che hanno bisogno di un "costruttore", implementano l'interfaccia **Initializable** di FXML utilizzando il metodo **initialize(...)** in ovveride. Quindi non viene usato un vero e proprio costruttore.

I dati vengono passati tramite il metodo **setData(...)**, chiamato durante la costruzione del controller. **initialize(...)** viene chiamato prima di **setData(...)**.

Il nome dei metodi di gestione di un pulsante (o elemento interattivo) iniziano con la parola **handle**.

3.4.1 FieldFormManagementController

File FXML associato: *nessuno*

Questa classe astratta si occupa di gestire i form di prenotazioni del campo, con i suoi metodi e ed i propri campi per ogni elemento della GUI.

3.4.2 BookFieldController

File FXML associato: *bookingForm*

Questa classe estende **FieldFormManagementController**, e gestisce il form di prenotazioni lato **User**, chiamando le varie classi in base al tipo di account ed ai propri permessi.

3.4.3 BookFieldManagerController

File FXML associato: *bookingFormManager*

Questa classe estende **BookFieldController**, e gestisce il form di prenotazioni lato **Manager**, chiamando le varie classi in base al tipo di account ed ai permessi da Manager.

3.4.4 BookFieldOwnerController

File FXML associato: *bookingFormOwner*

Questa classe estende semplicemente **BookFieldManagerController** dato che hanno i soliti permessi a riguardo, come mostrato in Figura 2.

3.4.5 FieldDetail

File FXML associato: *nessuno*

Questa classe astratta rappresenta la schermata che mostra i dettagli del campo da prenotare.

3.4.6 FieldDetailManagerController

File FXML associato: *fieldDetailManager*

Questa classe estende **FieldDetail** e mostra i dettagli del campo da prenotare lato **Manager**, con i relativi pulsanti dedicati.

3.4.7 FieldDetailUserController

File FXML associato: *fieldDetailsUser*

Questa classe estende **FieldDetail** e mostra i dettagli del campo da prenotare lato **User**.

3.4.8 FieldDetailOwnerController

File FXML associato: *fieldDetailOwner*

Questa classe estende **FieldDetail** e mostra i dettagli del campo da prenotare lato **Owner**, con i relativi pulsanti dedicati.

3.4.9 GroupItemController

File FXML associato: *groupItem*

Questa classe rappresenta un singolo item che rappresenta un **singolo** gruppo, che viene mostrato nella schermata che mostra tutti i gruppi (paragrafo 3.4.10).

3.4.10 YourGroupsController

File FXML associato: *groups*

Questa classe rappresenta la schermata dove sono presenti tutti i gruppi ai quale uno User fa parte. (singoli item al paragrafo 3.4.9)

3.4.11 InviteItemController

File FXML associato: *inviteItem*

Questa classe rappresenta un singolo item che rappresenta un **singolo** invito, che viene mostrato nella schermata che mostra tutti gli inviti lato User (paragrafo 3.4.12).

3.4.12 YourInvitesController

File FXML associato: *invites*

Questa classe rappresenta la schermata dove sono presenti tutti gli inviti ricevuti da uno User. (singoli item al paragrafo 3.4.11)

3.4.13 NotificationItem

File FXML associato: *nessuno*

Questa classe astratta rappresenta una **singola** notifica visualizzabile nella schermata delle notifiche ricevute (paragrafo 3.4.16).

3.4.14 NotificationItemController

File FXML associato: *notificationItem*

Questa classe estende solamente **NotificationItem** per renderla disponibile lato **User** e **Manager**.

3.4.15 NotificationItemOwnerController

File FXML associato: *notificationItemOwner*

Questa classe estende solamente **NotificationItem** per renderla disponibile lato **Owner**.

3.4.16 Notifications

File FXML associato: *nessuno*

Questa classe astratta rappresenta la schermata di notifiche ricevute dove sono presenti le singole notifiche (paragrafo 3.4.13).

3.4.17 NotificationsController

File FXML associato: *notifications*

Questa classe estende **Notifications** per renderla disponibile lato **User** e **Manager**.

3.4.18 NotificationsOwnerController

File FXML associato: *notificationsOwner*

Questa classe estende solamente **Notifications** per renderla disponibile lato **Owner**.

3.4.19 Menu

File FXML associato: *nessuno*

Questa classe astratta definisce campi e metodi comuni per i diversi tipi di menu.

3.4.20 MenuController

File FXML associato: *menuPane*

Questa classe eredita dalla **Menu** si occupa di gestire il menù ed i suoi pulsanti.

3.4.21 MenuOwnerController

File FXML associato: *menuPaneOwner*

Questa classe eredita dalla classe **Menu** e si occupa di gestire la *dashboard* dell'**Owner** ed i suoi pulsanti e funzioni dedicate.

3.4.22 ModifyReservationController

File FXML associato: *modifyReservation*

Questa classe eredita da **FieldFormManagementController** (paragrafo 3.4.1) e rappresenta il form di modifica di una prenotazione (lato User), come illustrato nei permessi di Figura 2.

3.4.23 ModifyReservationManagerController

File FXML associato: *modifyReservationManager*

Questa classe eredita da **ModifyReservationController** e rappresenta il form di modifica di una prenotazione forzata (lato Manager), come illustrato nei permessi di Figura 2.

3.4.24 ModifyReservationOwnerController

File FXML associato: *modifyReservationOwner*

Questa classe eredita da **ModifyReservationManagerController** e rappresenta il form di modifica di una prenotazione forzata (lato Manager), come illustrato nei permessi di Figura 2.

3.4.25 SelectGuestsPaneController

File FXML associato: *selectGuestsPane*

Questa classe rappresenta il *pop-up* (lato **User**) per gestire gli ospiti e gli inviti durante una prenotazione (aggiunta o modifica).

3.4.26 ManageGuestsManagerPaneController

File FXML associato: *editGuestsUserPane*

Questa classe rappresenta il *pop-up* (lato **Manager** e **Owner**) per gestire in modo forzato i membri di un gruppo (con i propri guests) relativo ad una prenotazione. Sia in fase di aggiunta che in fase di modifica.

3.4.27 ManageGuestsUserPaneController

File FXML associato: *editGuestsManagerOwnerPane*

Questa classe rappresenta il *pop-up* (lato **User**) per gestire gli ospiti, inviti ed una eventuale rimozione di membri (secondo i permessi in Figura 2) durante una prenotazione (aggiunta o modifica).

3.4.28 ManagementButtonsController

File FXML associato: *goToGroupButton* e *managementButton*

Questa classe gestisce due file FXML alternativi. Gestisce i pulsanti generati in altre schermate, come la schermata dei propri gruppi. Se la prenotazione a cui appartiene il gruppo è *matched* o non sono un capogruppo, verrà caricato il pulsante “*Go to groups*”. Se sono un capogruppo di una prenotazione *not matched*, verranno generati i pulsanti per modificare o rimuovere la prenotazione.

3.4.29 ReservationItemController

File FXML associato: *reservationItemUser*

Questa classe estende **ReservationItems** e rappresenta una **singola** prenotazione nella schermata delle prenotazioni di uno User (paragrafo 3.4.32).

3.4.30 ReservationItemManagerController

File FXML associato: *reservationItemManager*

Questa classe estende **ReservationItemsManagerOwner** e rappresenta una **singola** prenotazione nella schermata delle prenotazioni (lato **Manager**).

3.4.31 ReservationItemOwnerController

File FXML associato: *reservationItemOwner*

Questa classe estende **ReservationItemsManagerOwner** e rappresenta una **singola** prenotazione nella schermata delle prenotazioni (lato **Owner**).

3.4.32 ReservationsController

File FXML associato: *reservationsUser*

Questa classe rappresenta la schermata con tutte le prenotazioni di un utente, ciascuna gestita da un *ReservationItemController* (paragrafo 3.4.29).

3.4.33 SceneController

File FXML associato: *scene*

Questa classe gestisce la prima schermata, la quale porta automaticamente alla pagina di login.

3.4.34 AccessGui

File FXML associato: *nessuno*

Questa classe astratta si occupa di gestire le schermate di login e signUp e ne definisce le funzionalità.

3.4.35 Login

File FXML associato: *nessuno*

Questa classe astratta estende **AccessGui**. Gestisce le schermate di login definendo le funzioni invocabili dai file FXML.

3.4.36 LoginOwner

File FXML associato: *loginOwner*

Questa classe estende **Login**. Gestisce la schermata lato **Owner**.

3.4.37 LoginUser

File FXML associato: *loginUser*

Questa classe estende **Login**. Gestisce le schermata lato **User**.

3.4.38 SignUp

File FXML associato: *nessuno*

Questa classe astratta estende **AccessGui**. Gestisce le schermate di registrazione definendo i metodi comuni alle schermate degli **Owner** e degli **User**.

3.4.39 SignUpOwner

File FXML associato: *signUpOwner*

Questa classe estende **SignUp**. Gestisce le schermate di registrazione degli **Owner** implementando i metodi rimanenti.

3.4.40 SignUpUser

File FXML associato: *signUpUser*

Questa classe estende **SignUp**. Gestisce le schermate di registrazione degli **User** implementando i metodi rimanenti.

3.4.41 Announcement

File FXML associato: *nessuno*

Questa classe astratta gestisce la schermata per la creazione degli annunci definendo i metodi comuni alle schermate degli **Owner** e degli **User**.

3.4.42 AnnouncementOwner

File FXML associato: *announcementOwner*

Questa classe estende **Announcement**. Gestisce la schermata di creazione degli annunci degli **Owner** implementando il metodo **changeView()**.

3.4.43 AnnouncementManager

File FXML associato: *announcementManager*

Questa classe estende **Announcement**. Gestisce la schermata di creazione degli annunci dei **Manager** implementando il metodo **changeView()**.

3.4.44 MediaManagerController

File FXML associato: *nessuno*

Questa classe astratta gestisce il caricamento delle immagini da parte degli **implementando il metodo uploadImage()**.

3.4.45 Reservations

File FXML associato: *nessuno*

Questa classe astratta gestisce la schermata delle prenotazioni degli **Owner** e dei **Manager** implementando i metodi comuni e dichiarando i metodi astratti.

3.4.46 ReservationsManagerController

File FXML associato: *reservationsManager*

Questa classe estende **Reservations**. Gestisce la schermata delle prenotazioni dei **Manager** implementando i metodi rimanenti.

3.4.47 ReservationsOwnerController

File FXML associato: *reservationsOwner*

Questa classe estende **Reservations**. Gestisce la schermata delle prenotazioni degli **Owner**.

3.4.48 ReservationItems

File FXML associato: *nessuno*

Questa classe astratta rappresenta una **singola** prenotazione nella schermata delle prenotazioni.

3.4.49 ReservationItemsManagerOwner

File FXML associato: *nessuno*

Questa classe estende **ReservationItems**, rende la classe disponibile lato **Owner** e **Manager**.

3.4.50 ReservationItemOwnerController

File FXML associato: *reservationItemOwner*

Questa classe estende **ReservationItemsManagerOwner** e rappresenta una **singola** prenotazione nella schermata delle prenotazioni (lato **Owner**).

3.4.51 FacilityForm

File FXML associato: *nessuno*

Questa classe astratta estende **MediaManagerController**, essa gestisce le schermate di modifica e creazione degli impianti sportivi implementando i metodi comuni.

3.4.52 FieldForm

File FXML associato: *nessuno*

Questa classe astratta estende **MediaManagerController**. La classe gestisce le schermate di modifica e creazione dei campi sportivi implementando i metodi comuni e dichiarando il metodo astratto **newSport()**.

3.4.53 HomeOwnerController

File FXML associato: *homeOwner*

Questa classe gestisce la home degli **Owner**.

3.4.54 HomeUserController

File FXML associato: *homeUser*

Questa classe gestisce la home degli **User**.

3.4.55 UpdateProfileController

File FXML associato: *nessuno*

Questa classe astratta gestisce le schermate di modifica del profilo.

3.4.56 UpdateAddress

File FXML associato: *nessuno*

Questa classe astratta estende **UpdateProfileController**, gestisce la schermata di modifica dell'indirizzo.

3.4.57 UpdateAddressOwnerController

File FXML associato: *updateAddressOwner*

Questa classe estende **UpdateAddress**, rende disponibile la schermata lato **Owner**.

3.4.58 UpdateAddressUserController

File FXML associato: *updateAddressUser*

Questa classe estende **UpdateAddress**, rende disponibile la schermata lato **User** e **Manager**.

3.4.59 UpdateEmail

File FXML associato: *nessuno*

Questa classe astratta estende **UpdateProfileController**, gestisce la schermata di modifica dell'email.

3.4.60 UpdateEmailOwnerController

File FXML associato: *updateEmailOwner*

Questa classe estende **UpdateEmail**, rende disponibile la schermata lato **Owner**.

3.4.61 UpdateEmailUserController

File FXML associato: *updateEmailUser*

Questa classe estende **UpdateEmail**, rende disponibile la schermata lato **User** e **Manager**.

3.4.62 UpdatePassword

File FXML associato: *nessuno*

Questa classe astratta estende **UpdateProfileController**, gestisce la schermata di modifica della password.

3.4.63 UpdatePasswordOwnerController

File FXML associato: *updatePasswordOwner*

Questa classe estende **UpdatePassword**, rende disponibile la schermata lato **Owner**.

3.4.64 UpdatePasswordUserController

File FXML associato: *updatePasswordUser*

Questa classe estende **UpdatePassword**, rende disponibile la schermata lato **User** e **Manager**.

3.4.65 UpdateUsername

File FXML associato: *nessuno*

Questa classe astratta estende **UpdateProfileController**, gestisce la schermata di modifica dell'username.

3.4.66 UpdateUsernameOwnerController

File FXML associato: *updateUsernameOwner*

Questa classe estende **UpdateUsername**, rende disponibile la schermata lato **Owner**.

3.4.67 UpdateUsernameUserController

File FXML associato: *updateUsernameUser*

Questa classe estende **UpdateUsername**, rende disponibile la schermata lato **User** e **Manager**.

3.4.68 AddManagersController

File FXML associato: *addManagers*

Questa classe gestisce l'aggiunta di Manager ad un impianto sportivo da parte degli **Owner**.

3.4.69 ModifyWorkingHoursController

File FXML associato: *modifyWorkingHours*

Questa classe estende **MediaManagerController**, essa gestisce la schermata di modifica delle ore di apertura di un impianto sportivo.

3.4.70 ModifyFieldController

File FXML associato: *modifyField*

Questa classe estende **FieldForm**, essa gestisce la schermata di modifica di un campo sportivo.

3.4.71 ModifyFacilityController

File FXML associato: *modifyFacility*

Questa classe estende **FacilityForm**, essa gestisce la schermata di modifica di un impianto sportivo.

3.4.72 NewSportController

File FXML associato: *newSport*

Questa classe gestisce la schermata di creazione di uno sport.

3.4.73 NewFieldController

File FXML associato: *newField*

Questa classe estende **FieldForm**, gestisce la schermata di creazione di un campo sportivo.

3.4.74 NewFacilityController

File FXML associato: *newFacility*

Questa classe estende **FacilityForm**, gestisce la schermata di creazione di un impianto sportivo.

3.4.75 NewWorkingHoursController

File FXML associato: *newWorkingHours*

Questa classe gestisce la schermata di creazione delle ore di apertura di un impianto sportivo.

3.4.76 FieldChoice

File FXML associato: *nessuno*

Questa classe astratta gestisce le schermate di panoramica dei campi.

3.4.77 FieldChoiceOwnerController

File FXML associato: *fieldChoiceOwner*

Questa classe estende **FieldChoice**, rende disponibile la schermata lato **Owner**.

3.4.78 FieldChoiceManagerController

File FXML associato: *fieldChoiceManager*

Questa classe estende **FieldChoice**, rende disponibile la schermata lato **Manager**.

3.4.79 FieldItem

File FXML associato: *nessuno*

Questa classe astratta rappresenta un singolo campo sportivo visualizzabile nella panoramica dei campi.

3.4.80 FieldItemUserController

File FXML associato: *fieldItemUser*

Questa classe estende **FieldItem**, rende disponibile la classe lato **User**.

3.4.81 FieldChoiceItem

File FXML associato: *nessuno*

Questa classe astratta estende **FieldItem** per renderla disponibile per gli **Owner** e i **Manager**.

3.4.82 FieldChoiceItemOwnerController

File FXML associato: *fieldChoiceItemOwner*

Questa classe estende **FieldChoiceItem**, rende disponibile la classe lato **Owner**.

3.4.83 FieldChoiceItemManagerController

File FXML associato: *fieldChoiceItemManager*

Questa classe estende **FieldChoiceItem**, rende disponibile la classe lato **Manager**.

3.4.84 FacilityDetail

File FXML associato: *nessuno*

Questa classe astratta gestisce la schermata che mostra i dettagli di un impianto sportivo.

3.4.85 FacilityDetailManagerController

File FXML associato: *facilityDetailManager*

Questa classe estende **FacilityDetail**, rende disponibile la schermata lato **Manager**.

3.4.86 FacilityDetailOwnerController

File FXML associato: *facilityDetailManager*

Questa classe estende **FacilityDetail**, rende disponibile la schermata lato **Owner**.

3.4.87 FacilityChoice

File FXML associato: *nessuno*

Questa classe astratta gestisce le schermate di panoramica degli impianti sportivi.

3.4.88 FacilitiesController

File FXML associato: *facilitiesOwner*

Questa classe estende **FacilityChoice**, rende disponibile la classe lato **Owner** e permette la gestione degli impianti sportivi.

3.4.89 FacilityChoiceOwnerController

File FXML associato: *facilityChoiceOwner*

Questa classe estende **FacilityChoice**, rende disponibile la schermata lato **Owner**.

3.4.90 FacilityChoiceManagerController

File FXML associato: *facilityChoiceManager*

Questa classe estende **FacilityChoice**, rende disponibile la schermata lato **Manager**.

3.4.91 FacilityItem

File FXML associato: *nessuno*

Questa classe astratta rappresenta un singolo impianto sportivo visualizzabile nella panoramica dei campi.

3.4.92 FacilityItemController

File FXML associato: *facilityItem*

Questa classe estende **FacilityItem**, rende disponibile la classe lato **owner** e permette la gestione del singolo impianto sportivo.

3.4.93 FacilityChoiceItem

File FXML associato: *nessuno*

Questa classe astratta estende **FacilityItem** per renderla disponibile per gli **Owner** e i **Manager**.

3.4.94 FacilityChoiceItemOwnerController

File FXML associato: *facilityChoiceItemOwner*

Questa classe estende **FacilityChoiceItem**, rende disponibile la classe lato **Owner**.

3.4.95 FacilityChoiceManagerController

File FXML associato: *facilityChoiceItemManager*

Questa classe estende **FacilityChoiceItem**, rende disponibile la schermata lato **Manager**.

3.5 Database

Nel database sono presenti le tabelle elencate al paragrafo 2.7. Il database viene popolato con dei dati (file *default.sql*) di esempio rispettando i vincoli proposti dal modello.

3.5.1 Modelli

Le tabelle del database, ovvero i modelli vengono definire nel file *models.sql*.

Nello Snippet 24 viene mostrato il modello della tabella **Reservation**, alla riga 1, viene usato un comando utile per eliminare i vecchi dati per ricreare nuovamente la nuova tabella con il comando **CREATE TABLE**:

```
1 DROP TABLE IF EXISTS "Reservation" CASCADE;
2
3
4 CREATE TABLE IF NOT EXISTS "Reservation" (
5     id SERIAL PRIMARY KEY,
6     res_date DATE NOT NULL DEFAULT CURRENT_DATE,
7     event_date DATE NOT NULL,
8     res_time TIME NOT NULL DEFAULT CURRENT_TIME,
9     event_time_start TIME NOT NULL,
10    event_time_end TIME NOT NULL,
11    id_field INTEGER NOT NULL,
12    is_confirmed BOOLEAN NOT NULL DEFAULT FALSE,
13    is_matched BOOLEAN NOT NULL DEFAULT FALSE,
14    is_deleted BOOLEAN NOT NULL DEFAULT FALSE,
15    is_notified BOOLEAN NOT NULL DEFAULT FALSE,
16    FOREIGN KEY (id_field) REFERENCES "Field"(id) ON DELETE CASCADE ON UPDATE
17    CASCADE
18 );
```

Snippet 24: Implementazione del modello della tabella Reservation presente nel file *models.sql* per il database.

L'utilizzo dei comandi **DROP TABLE** e **CREATE TABLE**, come descritti prima, vengono usati per tutte le tabelle presenti del sistema.

3.5.2 Triggers

Il database utilizza dei triggers definiti nel file *triggers.sql*.

Infatti i trigger definiscono delle regole per mantenere coerenza e pulizia tra i dati di alcune tabelle. Queste regole sono definite all'interno di **funzioni di trigger**, successivamente utilizzate dai trigger stessi.

- **delete_unused_messages()**, Snippet 26: cancella automaticamente un messaggio dalla tabella **Message** se non è più associato a nessuna notifica (**NotificationUser** o **NotificationOwner**).
- **fn_check1_and_delete_reservation()**, Snippet 27: elimina una prenotazione della tabella **Reservation** che ha la flag **is_deleted = true** e non è associata a nessuna notifica in **NotificationUser** o **NotificationOwner**. Questa funzione di trigger è molto importante perché è l'unico modo per eliminare concretamente una prenotazione dal database, dato che da Java viene solamente asserita la flag *is_deleted*.
- **cleanup_old_reservations()**, Snippet 28: cancella automaticamente tutte le prenotazioni che hanno la flag **is_deleted = true** e sono di almeno di 31 giorni fa rispetto alla data odierna.

Nello snippet 25 sono mostrati i comandi per eliminare eventuali trigger preesistenti, evitando conflitti durante la ridefinizione.

```

1 -- drop triggers
2 DROP TRIGGER IF EXISTS trg_check_delete_reservation_user ON "NotificationUser";
3 DROP TRIGGER IF EXISTS trg_check_delete_reservation_owner ON "NotificationOwner"
4 ;
5 DROP TRIGGER IF EXISTS trigger_delete_message_user ON "NotificationUser";
6 DROP TRIGGER IF EXISTS trigger_delete_message_owner ON "NotificationOwner";
7 DROP TRIGGER IF EXISTS trg_cleanup_old_reservations ON "Reservation";

```

Snippet 25: Implementazione dei comandi *DROP TABLE* per eliminare eventuali versioni preesistenti dei trigger.

```

1 CREATE OR REPLACE FUNCTION delete_unused_messages()
2 RETURNS TRIGGER AS $$$
3 BEGIN
4     -- checks if message has some references
5     IF OLD.id_message IS NOT NULL THEN
6         IF NOT EXISTS (
7             SELECT 1
8                 FROM "NotificationUser"
9                 WHERE id_message = OLD.id_message
10        ) AND NOT EXISTS (
11            SELECT 1
12                FROM "NotificationOwner"
13                WHERE id_message = OLD.id_message
14        ) THEN
15            -- delete message if there aren't references
16            DELETE FROM "Message"
17            WHERE id = OLD.id_message;
18        END IF;
19    END IF;
20
21    RETURN NULL;
22 END;
23 $$ LANGUAGE plpgsql;

```

Snippet 26: Implementazione della funzione di trigger **delete_unused_messages()**

```

1 CREATE OR REPLACE FUNCTION fn_check_and_delete_reservation()
2 RETURNS trigger AS $$$
3 DECLARE
4     reservation_id INTEGER;
5     reservation_status BOOLEAN;
6 BEGIN
7     -- takes reservation id by deleted record
8     reservation_id := OLD.id_reservation;
9
10    IF reservation_id IS NOT NULL THEN
11        -- check reservation state (is_deleted)
12        SELECT is_deleted INTO reservation_status
13        FROM "Reservation"
14        WHERE id = reservation_id;
15
16        -- if reservation is marked as deleted...
17        IF reservation_status = true THEN
18            -- ... and there aren't any references...
19            IF NOT EXISTS (
20                SELECT 1 FROM "NotificationUser" WHERE id_reservation =
21                reservation_id
22            ) AND NOT EXISTS (

```

```

22         SELECT 1 FROM "NotificationOwner" WHERE id_reservation =
23             reservation_id
24     ) THEN
25 DELETE FROM "Reservation" WHERE id = reservation_id;
26 END IF;
27 END IF;
28
29 RETURN OLD;
30 END;
31 $$ LANGUAGE plpgsql;

```

Snippet 27: Implementazione della funzione di trigger `fn_check_and_delete_reservation()`

```

1 CREATE OR REPLACE FUNCTION cleanup_old_reservations()
2 RETURNS TRIGGER AS $$$
3 BEGIN
4 DELETE FROM "Reservation"
5 WHERE is_deleted = true
6   AND event_date < CURRENT_DATE - INTERVAL '31 days';
7
8 RETURN NEW;
9 END;
10 $$ LANGUAGE plpgsql;

```

Snippet 28: Implementazione della funzione di trigger `cleanup_old_reservations()`

I trigger vengono creati per tutte le tabelle interessate (ad esempio, **NotificationUser** e **NotificationOwner**) e per ogni funzione di trigger, come mostrato nello snippet 29.

```

1 CREATE TRIGGER trigger_delete_message_user
2   AFTER DELETE
3   ON "NotificationUser"
4   FOR EACH ROW
5   EXECUTE FUNCTION delete_unused_messages();

```

Snippet 29: Esempio di implementazione di creazione di un trigger, utilizzando una funzione di trigger.

Invece il trigger che utilizza la funzione `cleanup_old_reservations()`, ovvero `trg_cleanup_old_reservations`, viene eseguito ad ogni *INSERT* o *UPDATE* di un'istanza in **Reservation**, eseguendo la funzione per *statement* e non per ogni riga.

3.5.3 Transactions

Le transactions garantiscono l'integrità dei dati durante operazioni complesse, soprattutto quando si modificano più tabelle in un'unica operazione.

In questo sistema, il **primo chiamante dei DAO** gestisce ogni transazione, costruendo l'operazione desiderata. I **rollback** sono gestiti nei blocchi **catch** delle eccezioni lanciate da fallimenti nella transazione.

L'**inizio** e il **commit** della transazione avvengono nel blocco **try**, mentre la fine è gestita dai blocchi **finally**.

Se le transaction coinvolgono anche il DomainModel, deve essere eseguita in parallelo per garantire coerenza con il database. Per farlo è necessario fare delle deep copy su tutti gli oggetti coinvolti, nello **stesso livello** in cui viene gestita la transaction. Dopo aver fatto il commit su database, è possibile applicare anche le modifiche effettuate su DomainModel.

Infatti le transazioni sono state gestite nei metodi delle BusinessLogic che ne hanno bisogno. Se utilizzano metodi in chiamate a catena, le eccezioni vengono propagate fino al primo chiamante dove viene gestita la transazione, un esempio è mostrato nello Snippet 8.

I metodi della BusinessLogic che usano transazioni sono:

- **addReservation(...)** della classe **PersonController**
- **editReservation(...)** della classe **PersonController**
- **editWorkingHours(...)** della classe **OwnerManagementController**
- **acceptInvite(...)** della classe **UserActionsController**
- **leaveGroup(...)** della classe **UserActionsController**

Tutti questi metodi eseguono almeno due chiamate a DAO per aggiungere, modificare o eliminare istanze su diverse tabelle. Per garantire l'integrità dei dati in caso di fallimenti, sono necessarie delle transazioni, poiché l'operazione è considerata valida solo se tutte le chiamate DAO vanno a buon fine.

Il metodo **addNotification(...)** di NotificationDAO utilizza una transaction per l'aggiunta del annuncio, poiché interagisce con le tabelle **Message** e **NotificationUser/NotificationOwner**.

4 Testing

Sono stati eseguiti test sui principali packages per verificarne il corretto funzionamento. I metodi della BusinessLogic e dell'ORM sono stati sottoposti a test funzionali, mentre il Domain Model a test strutturali.

Ogni package contiene una classe **GeneralTest** (per il Domain Model) dalla quale ereditano **tutte** le classi di test. Per i test di ORM e BusinessLogic, è necessario ricreare un ambiente isolato simile a quello del client. Le classi **GeneralBTest** e **GeneralDAOTest** forniscono, dove necessario, i metodi astratti **setup()** e **teardown()** per l'impostazione e il rilascio delle risorse per **ogni** test. Un esempio per la BusinessLogic è mostrato in Figura 55.

La classe **GeneralTest** include metodi per creare rapidamente oggetti utili nei test, come mostrato nello Snippet 32.

4.1 DomainModelTest

Per i test del Domain Model **NON** sono necessari i metodi **setup()** e **teardown()** nella classe **GeneralTest**.

Sono state testate solamente le classi del DomainModel che avessero rilevanza o logiche complesse.

4.1.1 GeneralTest

Questa classe astratta è la base per tutte le altre classi di **DomainModelTest**. Permette di creare rapidamente oggetti per i test, come nell'esempio mostrato nello Snippet 32.

4.1.2 GroupTest

Questa classe testa la classe **Group** del Domain Model, testando anche l'inizializzazione tramite il costruttore (con **initGroupTest()**) e i metodi chiave per il corretto funzionamento della classe (**addMemberTest(...)**, **removeMemberTest(...)**, **changeUserGuestsTest(...)**, **requiredParticipantsTest()**, **canJoinTest(...)**).

Per isolare i test, dato che Group dipende dall'observer tramite la sua Reservation, il metodo privato **disableObserver(...)** lo disabilita temporaneamente per il test.

4.1.3 ReservationTest

Questa classe testa la classe **Reservation** del Domain Model, testando anche l'inizializzazione tramite il costruttore (con **initReservationTest()**) e i metodi chiave per il corretto funzionamento della classe (**isTimeOverlappingTest()** e **isEndTimeAfterStartTimeTest()**).

4.1.4 FacilityTest

Questa classe testa la classe **Facility** del Domain Model, testando anche l'inizializzazione tramite il costruttore (con **initReservationTest()**).

4.1.5 InviteSenderTest

Essa si occupa di testare la classe **InviteSender**, in particolar modo il metodo **factoryMethod()** controllando che l'invito restituito non sia nullo e corrisponda al gruppo giusto.

4.2 BusinessLogicTest

Siccome le Business Logic interagiscono col database come mostrato in Figura 1, per garantire dei test isolati, è necessario rimuovere temporaneamente la dipendenza del database attraverso dei **mock**. Essi sono oggetti "finti" utili per simulare il comportamento di oggetti reali con la possibilità di controllarne il funzionamento.

Per assegnare i mock (creati nel metodo di **setup()** della classe di test, mostrato nello Snippet 30) dei DAO alle Business Logic costruite in fase di test è necessario un **overload** dei costruttori in ogni Business Logic (non di test) dedicato ai relativi test. Questi costruttori assegnano tutti i parametri ai campi di classe in modo tale da "controllarli" dall'esterno, ovvero attraverso i mock creati, come mostrato nello Snippet 31.

```

1  @Override
2  @BeforeEach
3  public void setup() throws SQLException {
4
5      person = createUser();
6
7      personDAOMock = mock(UserDAO.class);
8
9      profileController = new UserProfileController(person, personDAOMock);
10 }
11
12 @Override
13 @AfterEach
14 public void teardown() {
15
16     person = null;
17     personDAOMock = null;
18     profileController = null;
19
20 }

```

Snippet 30: Esempio dei metodi **setup()** e **teardown()** della classe **UserProfileControllerTest** del package **BusinessLogicTest**

```

1  public NotificationController(Person person, FacilityDAO facilityDAO,
2     OwnerDAO ownerDAO, NotificationDAO notificationDAO, IsPartDAO isPartDao,
3     ManagesDAO managesDAO, GroupDAO groupDAO, ReservationDAO reservationDao) {
4     this.person = person;
5
6     this.facilityDAO = facilityDAO;
7     this.ownerDAO = ownerDAO;
8     this.notificationDAO = notificationDAO;
9     this.isPartDao = isPartDao;
10    this.managesDAO = managesDAO;
11    this.groupDAO = groupDAO;
12    this.reservationDao = reservationDao;
13 }
14

```

Snippet 31: Esempio di un overload del costruttore per **NotificationController**. Classe non di test.

Per i test della Business Logic, la classe **GeneralTest** richiede i metodi astratti **setup()** e **teardown()** per isolare e resettare **ogni** test.

In Figura 55 è mostrato un esempio della catena di ereditarietà delle classi di test all'interno del package **BusinessLogicTest**. Nella classe GeneralBTest sono presenti dei metodi che restituiscono, come quello mostrato nello Snippet 32.

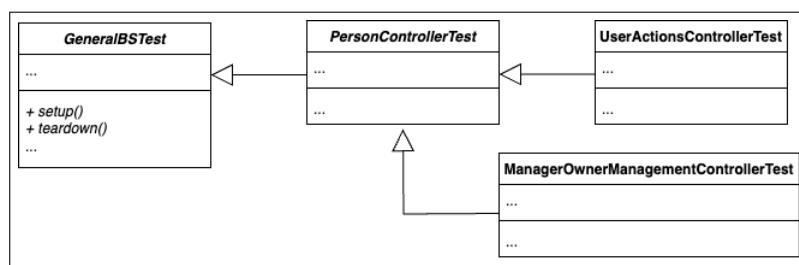


Figura 55: Esempio di Class Diagram delle classi di alcune **classi di test** della BusinessLogic. In particolare notare l'uso di **GeneralBTest**.

4.2.1 GeneralBTest

Questa classe astratta è la base per tutte le altre classi di **BusinessLogicTest**. Permette di creare rapidamente oggetti per i test, come nell'esempio mostrato nello Snippet 32.

```

1  protected User createUser(int userNumber) {
2      return new User(userNumber, "hello"+userNumber+"@gmail.com", "user"+
3          userNumber, "hello123", "London", "London", "00000", "UK");
4  }

```

Snippet 32: Implementazione del metodo **createUser(...)** della classe **GeneralBTest**.

4.2.2 NotificationControllerTest

Questa classe testa **NotificationController**, quindi l'invio delle notifiche e il corretto funzionamento dell'observer. I test includono **sendConfirmNotificationTest()**, **updateObserverTest()**, ecc. L'observer viene testato tramite il suo metodo **update()**, che gestisce gli aggiornamenti notificati.

4.2.3 PersonControllerTest

Questa classe astratta testa **PersonController** della Business Logic, racchiudendo anche dei mock dei DAO e dei metodi protetti (macro) con oggetti finti di Mockito, usati in diversi test.

Vengono testati tutti i metodi fondamentali del sistema di **PersonController**, quelli a che si interfacciano con tutti i fruitori, come le prenotazioni ed la gestione dei gruppi (secondo i propri permessi). Alcuni di questi sono **editReservationTest()**, **deleteReservationTest()**, **joinGroupHelperTest()**, **removeGroupMembersTest()**, ecc...

L'implementazione dei metodi **setup()** e **teardown()** viene delegata alle classi figlie (non astratte).

Nonostante questo, viene ripresa la catena di ereditarietà di **PersonController** anche nelle classi di test relative.

4.2.4 UserActionControllerTest

Questa classe estende **PersonControllerTest** e testa i metodi più importanti di **UserActionsController** della Business Logic, come la gestione degli inviti (accetta o declina) o lasciare/unirsi ad un gruppo. Alcuni di questi metodi sono **acceptInviteTest()**, **declineInvite()**, **leaveGroupTest()**, ecc...

Implementa anche il metodo astratto **setup()** (e **teardown()**) per assegnare i **mock** ai costruttori delle Business Logic. Questo permette di testare tutti i DAO isolando i test dal database.

4.2.5 ProfileController

Questa classe astratta testa **ProfileController**, un'altra classe astratta che usa i Java Generics. Come **ProfileController**, **ProfileControllerTest** usa i Java Generics per stabilire a runtime il tipo di **Person** e **PersonDAO**, come mostrato nello Snippet 33. Include anche un metodo astratto (**createPerson()**) per creare dinamicamente il giusto tipo di persona, come mostrato nello Snippet 34.

Questa classe contiene i test per tutti i metodi di **ProfileController** della Business Logic, utili a effettuare modifiche sul proprio account. Ad esempio **updateEmailTest()**, **updateUsernameTest()**, ecc...

Le implementazioni dei metodi di **setup()** e **teardown()**, ereditati da **GeneralBTest**, vengono **delegate** alle classi che la ereditano.

```
1 public abstract class ProfileControllerTest<T extends Person, D extends PersonDAO> extends GeneralBTest {
2
3     protected T person;
4
5     protected D personDAOMock;
6
7     protected ProfileController profileController;
8
9     protected abstract T createPerson();
10
11    //tests
12    //...
13    //...
```

Snippet 33: Pezzo di implementazione della classe **PersonControllerTest**

4.2.6 UserProfileController

Questa classe estende **ProfileControllerTest**, passando **User** e **UserDAO** come Generics. L'override del metodo **createPerson()** mostrato nello Snippet 34 restituisce il tipo **User**, chiamando **createUser()** di **GeneralBTest**. Non contiene ulteriori test, infatti vengono ereditati tutti.

```
1     @Override
2     protected User createPerson() {
3         return createUser();
4     }
```

Snippet 34: Implementazione override di **createPerson()** della classe **PersonControllerTest**

4.2.7 OwnerProfileController

Analogamente a **UserProfileController**, anche questa classe estende **ProfileControllerTest**, passando **Owner** e **OwnerDAO** come Generics. L'override del metodo **createPerson()** mostrato nello Snippet 34 restituisce il tipo **Owner**, chiamando **createOwner()** di **GeneralBTest**. Non contiene ulteriori test, infatti vengono ereditati tutti.

4.2.8 OwnerManagementControllerTest

Questa classe si occupa di testare **OwnerManagementController**, che estende **ManagerOwnerManagementControllerTest**, testa sia le varie funzioni statistiche sia le funzionalità esclusive dell' **Owner** come per esempio **attachManager()** e **detachManager()** o le varie funzioni di creazione, modifica e cancellazione di campi e impianti sportivi controllando che restituiscano il risultato previsto e che lancino le dovute eccezioni quando necessario.

4.2.9 ManagerOwnerManagementControllerTest

La classe interessata dal test è **ManagerOwnerManagementController**, che estende **PersonControllerTest**, sono interessate dal test quindi le funzionalità condivise del **Manager** e dell' **Owner**. Per esempio **reservationAnnouncement(...)** e **getFieldsByFacility()**.

4.2.10 AccessOwnerTest

Essa si occupa di testare la classe **AccessController** lato **Owner**, in particolare sono di particolare interesse i metodi **login** e **register**. Lo fa controllando che tutti i metodi della classe restituiscano il risultato previsto e che lancino le dovute eccezioni quando necessario.

4.2.11 AccessUserTest

Essa si occupa di testare la classe **AccessController** lato **User**, anche in questo caso sono di particolare interesse i metodi **login** e **register**.

4.3 ORMTest

A differenza degli altri package di test, l'ORM richiede l'esecuzione di azioni su database per testare i DAO.

La classe **GeneralDAOTest** definisce i metodi astratti **setup()** e **teardown()** per creare e distruggere istanze fittizie, come descritto al paragrafo 4.3.1.

Vengono usate le **Assumptions** di Junit per gestire gli errori durante l'aggiunta o la rimozione di istanze fittizie, inibendo i test successivi.

Questo approccio consente di risolvere manualmente gli errori agendo su un numero limitato di istanze.

I metodi di aggiunta e rimozione vengono “testati” direttamente nei metodi **setup()** e **teardown()**, come mostrato nello Snippet 35.

All'inizio di **ogni** test, un'Assumption verifica se saltare il test in base al campo statico booleano **shouldSkip**, impostato a *true* nei metodi **setup()** o **teardown()**.

Le classi **ConnectionManager** e **ConnectionHolder** sono testate indirettamente tramite altri DAO. Se non funzionano, il test fallirebbe prima.

```
1  @Override
2  @BeforeEach
3  public void setup() throws SQLException {
4      personDAO = new UserDAO();
5
6      person = createUser();
7
8      if (person.getId() == 0)
9          shouldSkip = true;
10
11 }
12
13 @Override
14 @AfterEach
15 public void teardown() throws SQLException {
16
17     if (person != null && person.getId() != 0)
```

```

18         personDAO.deletePerson(person.getUsername());
19
20     person = null;
21     personDAO = null;
22
23     //it is important to reset each test
24     shouldSkip = false;
25 }
```

Snippet 35: Metodi **setup()** e **teardown()** della classe **UserDAOTest** del package **ORMTest**

4.3.1 GeneralDAOTest

GeneralDAOTest, come altre classi di test, crea oggetti “prefabbricati” (**createUser()**, **createFacility()**, ecc...) e possiede i metodi astratti **setup()** e **teardown()**. A differenza delle altre, questi metodi aggiungono istanze al database, poi cancellate con i **teardown()**. Imposta anche l’id generato dal database. In questo caso, i metodi di creazione degli oggetti sono pensati per essere chiamati **soltamente** nei **setup()** per distruggerli nei **teardown()**. Questa classe verrà estesa da **tutte** le altre appartenenti a questo package, come mostrato in Figura 55 per **BusinessLogicTest**.

4.3.2 FacilityDAOTest

In questa classe vengono testati tutti i metodi di **FacilityDAO** dell’ORM, ovvero quelli di *update*, come ad esempio **updateCountry()**.

4.3.3 ManagesDAOTest

In questa classe vengono testati tutti i metodi di **ManagesDAO** dell’ORM, ovvero quelli di gestione dei manager, come ad esempio **attachManager()** o **detachManager()**.

4.3.4 PersonDAOTest

In questa classe vengono testati tutti i metodi di **PersonDAO** dell’ORM, ovvero quelli di *update* del profilo e alcuni DAO di *get*, come ad esempio **updateCountry()** o **getUser()**.

Per permettere alle classi figlie (**UserDAOTest** e **OwnerDAOTest**) di ereditare tutti i test accedendo alle corrette tabelle di database, vengono utilizzati i Java Generics. Viene definito anche un metodo astratto (**getPerson()**) che permette di chiamare direttamente **getUser(...)** o **getOwner(...)** dei relativi DAO, bypassando i generics. Viene mostrato un esempio di implementazione nello Snippet 36.

```

1 public abstract class PersonDAOTest<T extends Person, D extends PersonDAO>
2     extends GeneralDAOTest {
3
4     protected D personDAO;
5     protected T person = null;
6     protected static boolean shouldSkip = false;
7
8     protected abstract T getPerson(String username) throws SQLException;
9
10    //...
11    //...
```

Snippet 36: Frammento dell’implementazione di **PersonDAOTest**

4.3.5 OwnerDAOTest

Questa classe testa tutti i metodi di **OwnerDAO** dell’ORM, come **addOwner()** o **getOwner()**, oltre a ereditare i test di **PersonDAOTest** utilizzando le tabelle del DB relative all’Owner.

4.3.6 UserDAOTest

Questa classe testa tutti i metodi di **UserDAO** dell’ORM, inclusi quelli di *get* come **getUsersByProvinceTest()**, ereditando i test di **PersonDAOTest** con le tabelle DB relative all’User.

4.3.7 WorkingHoursDAOTest

Questa classe testa tutti i metodi di **WorkinHoursDAO** dell'ORM, ovvero i metodi di *get* come **getWHsByFacility()**, **getWHsByFacilityByDay()** o **getWH()**.

4.3.8 FieldDAOTest

Questa classe si occupa di testare **GroupDAO**, testa le varie funzioni di modifica e di ricerca come **updateImage(...)** e **getFieldsByOwner(...)** e funzioni statistiche come **reservedFields(...)**.

4.3.9 ReservationDAOTest

Questa classe si occupa di testare **ReservationDAO** e i suoi metodi. Ci sono metodi di modifica, statisticci e di ricerca come **dailyEarning(...)** e **updateEventDate(...)** e **getReservationsByField(...)**.

4.3.10 GroupDAOTest

Essa testa la classe **GroupDAO** e i suoi metodi come per esempio **getGroup(...)**.

4.3.11 InviteDAOTest

Questa classe si occupa di testare **InviteDAO** e le sue funzioni come per esempio **getInvitesByUser(...)**.

4.3.12 IsPartDAOTest

Questa classe si occupa di testare **IsPartDAO** e le sue funzioni come **countOwnGuests(...)** e **getAllGroupsByUser(...)** controllando che il risultato sia quello atteso.

4.3.13 NotificationDAOTest

Essa testa la classe **NotificationDAO** e i metodi **getNotification(...)** e **getNotifications(...)**.

4.3.14 SportDAOTest

Essa testa la classe **SportDAO**, tra i metodi testati ci sono i metodi di modifica e di ricerca come **updateSportPlayers(...)** e **getAllSport()**.

5 Bibliografia

Tutto il codice è stato scritto manualmente dagli sviluppatori.

5.1 Utilizzo di AI

Tutte le tabelle del database sono state popolate con l'aiuto di OpenAI ChatGPT.