Bilkent University

Department of Computer Engineering

Senior Design Project

Project short-name: Coupl

Low-level Design Report

Cankat Anday Kadim, Rüzgar Ayan, Ege Türker, Emre Derman, Kamil Kaan Erkan

Supervisor: Cevdet Aykanat
Jury Members: Erhan Dolak and Tağmaç Topal

Low-level Design Report

February 28, 2022

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.

Contents

1 Introduction

Coupl is a mobile application that will create a safe environment for users to meet with new people in-person only in a matter of minutes. Event organizers will make their events available to the application and the users will join the events by scanning the QR codes provided in these event locations. The possible events include musical concerts, parties, art exhibitions and many more. Coupl will then match the users in the same event, based on their preference from the current participant pool and their history of event and matches in the application.

Coupl as a system/service aims to be an innovative alternative to the currently used dating and meeting services. By matching the users that are already participating in the same event and also with similar histories/profiles, Coupl aims to match the users of very similar interests/tastes. Compared to alternatives such as online swipe-based dating apps which only provide chatting functionality, this application will provide a much more dynamic experience with an in-person meeting.

1.1 Object design trade-offs

The object design trade-offs will be examined in this section, which compare the application's efficiency,concurrency, feasibility, extensibility, scalability, usability.

1.1.1 Efficiency vs Concurrency

Coupl's most essential qualities are efficiency and concurrency. When it comes to trade-offs, though, the value of concurrency trumps efficiency.
Concurrency of the program is critical in the Concurrency of the application should be long-term and not cause any issues or bugs in the system. The efficiency feature will be useless if it isn't sustainable enough.

1.1.2 Feasibility vs Extensibility

Apps' earliest versions are typically demos, and they are changed multiple times after their first release. The amount of data transferred in the program rises with each capacity expansion, improving the user experience. As a result, our application's extensibility is critical. The practicality of the Coupl's is also crucial. More essential to Coupl's than practicality is offering a long-term service and improving application extensibility.

1.1.3 Scalability vs Efficiency

The scalability of the Coupl' is one of its most important aspects. Although it is crucial to reach as many consumers as possible, the more important goal is to provide them with a reliable and effective service. As a result, while scalability is one of the application's essential features, efficiency is more important in terms of boosting the application's usability.

Admin
There should be a lot of capability in the front-end for the admin side to manage all processes linked to customer orders. Because this endpoint is mostly used for data modification, the user interface should be more functional rather than more usable. Thus, while the admin side's interface may be more difficult to master, it offers a wealth of functions for modifying and managing all orders-related transactions.

User
In the front-end for the user side, the main function of the application should be to create accounts, handle the matching case and send them to the back-end server. Therefore, there is not much functionality needed in the user interface, instead a more usable one should be prefered.

1.2 Interface documentation guidelines

In this report, all class names are typed singular and named ‘ClassName’. Variables and methods are named in the camel case as in ‘variableName’, ’methodName()’. The following table template is used for the class interfaces:

Class:	**ClassName**
General class description.	
Attributes:	
attribute_1: type	attribute_1’s description
attribute_2: type	attribute_2’s description
Methods:	
method_1(params): return_type	method_1’s description
method_2(params): return_type	method_2’s description

Table 1: Class Interface Template

1.3 Engineering standards

UML guidelines[1] will be followed strictly for the classes, diagrams, scenarios and use cases, subsystem compositions. IEEE engineering standards are used for the design of the project.

1.4 Definitions, acronyms, and abbreviations

- **API**: Application Programming Interface.
- **REST** (or **REST API**): Representational state transfer, an architectural style used in the development of web services.
- **UI**: User Interface.
- **HTTP**: Hypertext Transfer Protocol.
- **QR Code:** Quick Response Code

2 Packages

2.1 Client Side

2.1.1 User

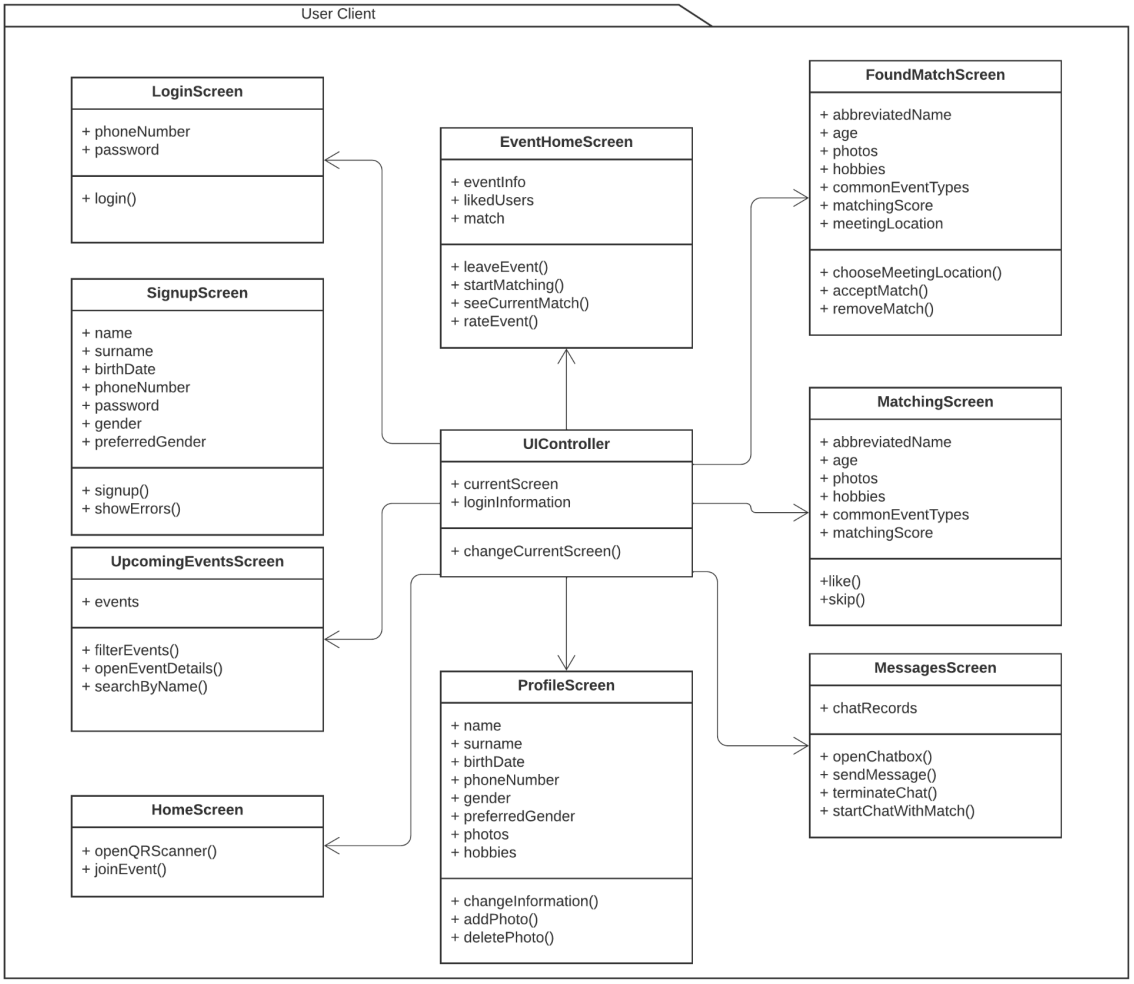

Figure 1: User Client Package

2.1.1.1 UIController

Controls navigation between screens.

2.1.1.2 LoginScreen

Login screen for users to login to the app.

2.1.1.3 SignupScreen

Sign-up screen for users to sign up for the app.

2.1.1.4 UpcomingEventsScreen

Upcoming events screen for users to get information about the upcoming events.

2.1.1.5 HomeScreen

Home screen for users to scan their QR codes to attend events.

2.1.1.6 ProfileScreen

Profile screen for users where users can see and edit their profile information.

2.1.1.7 MessagesScreen

Message screen for users to message with their matches.

2.1.1.8 MatchingScreen

Matching screen for users to see other users in the event.

2.1.1.9 FoundMatchScreen

Match screen for users to see their current match in the event.

2.1.1.10 EventHomeScreen

Event home screen for users to see their matching activity in an event.

2.1.2 Event Organizer

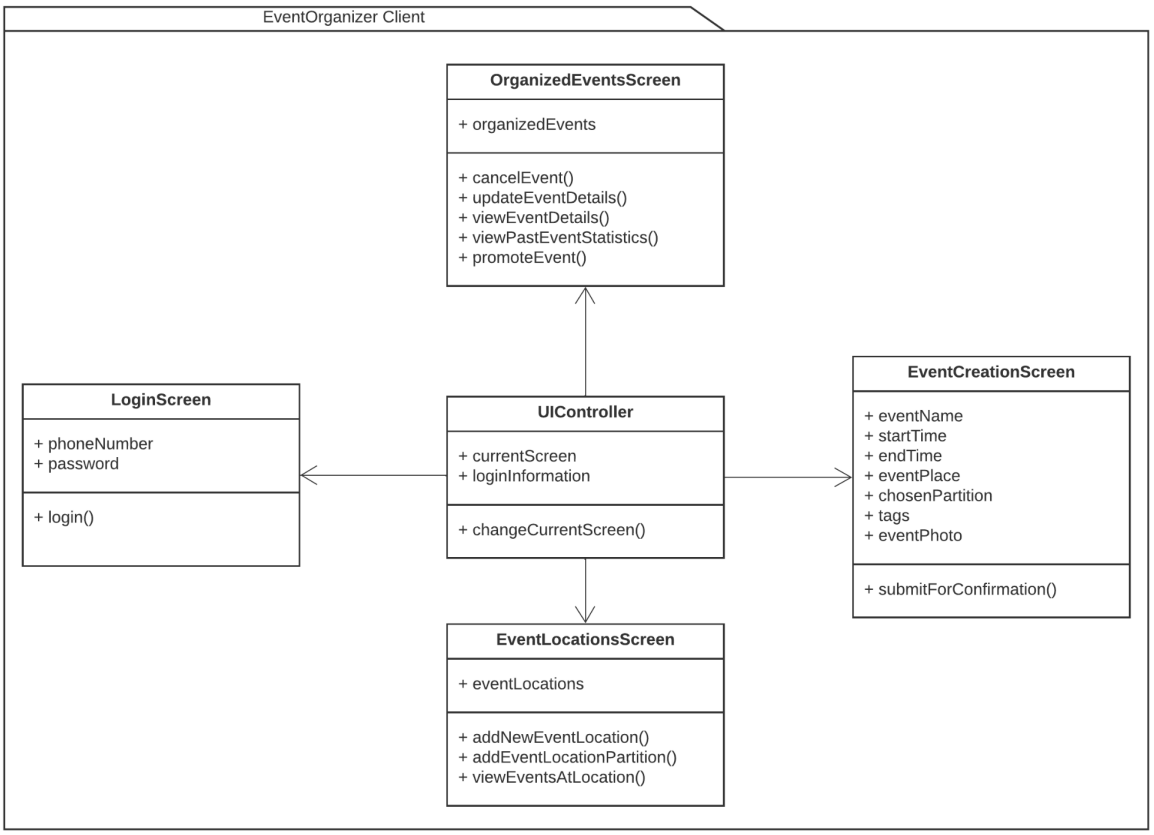

Figure 2: Event Organizer Client Package

2.1.2.1 UIController

Controls navigation between screens.

2.1.2.2 LoginScreen

Login screen for event organizers to login to the app.

2.1.2.3 OrganizedEventsScreen

The main screen where the event organizers see all their past and future events.

2.1.2.4 EventCreationScreen

Event creation screen where event organizers can schedule a new event.

2.1.2.5 EventLocationsScreen

Event location screen where event organizers can add new event locations and new event location partitions.

2.2 Server Side

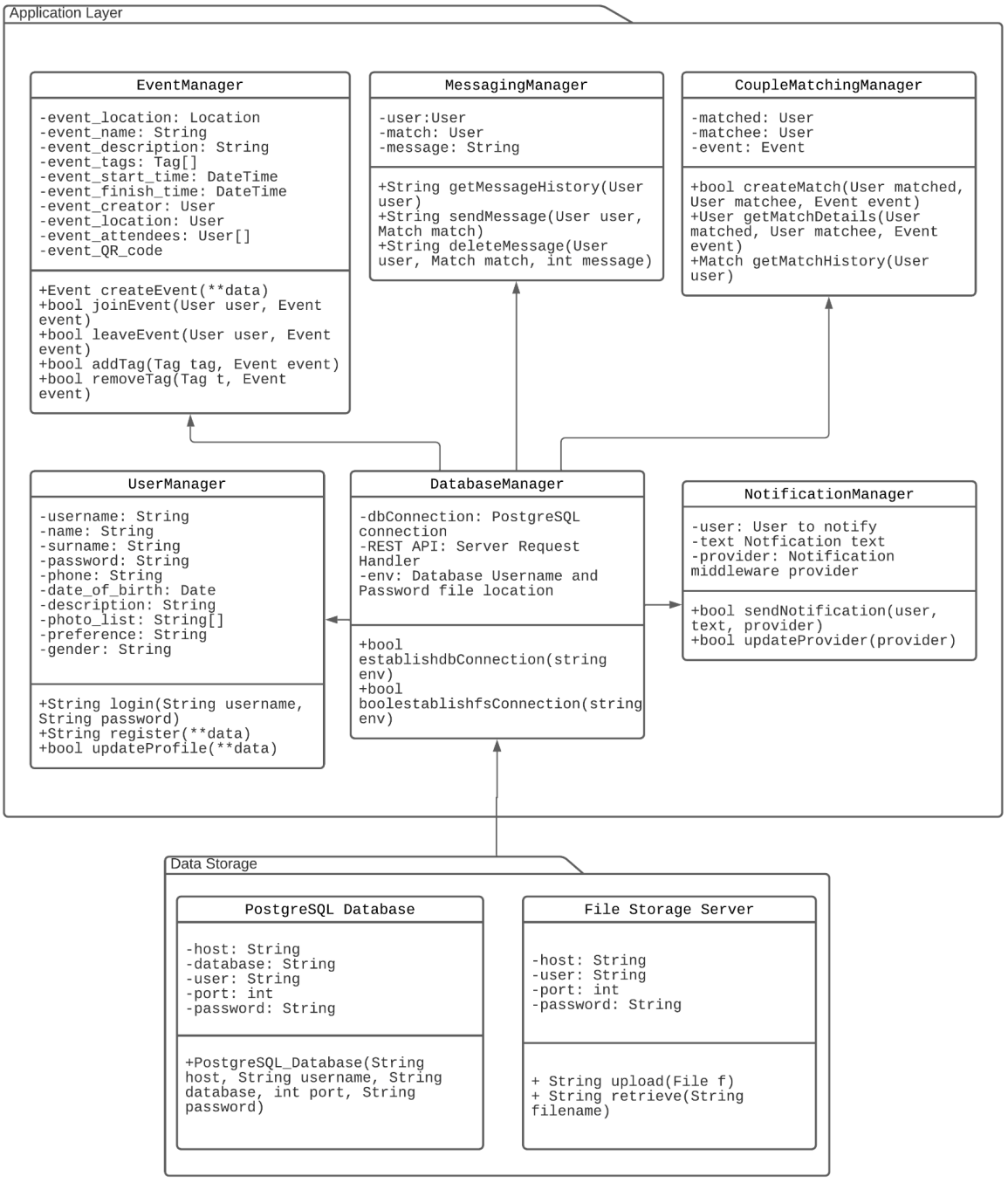

Figure 3: Server Side Packages

2.2.1 Couple Matching Manager

This module manages the user to user interactions and manages the previous and future matches in any event with both users attending the event.

2.2.2 Notification Manager

This module manages the notification backend of the service and notifies users when the event has started, when they have matched with another user or when they have made any modifications to their profile.

2.2.3 User Manager

This module manages the user's interaction with the service, as per login and signup. This manager also handles the profile and the data kept in the database regarding the individual user.

2.2.4 Database Manager

This module establishes connections to the file storage server and the PostgreSQL database. Through the REST API, the database manager handles the requests and CRUD operations.

2.2.5 Event Manager

This module manages the creation of events and the join/leave operations of the users to those events.

2.2.6 Messaging Manager

This module manages the direct messaging between a user and their match. Users can send, delete messages or view their message history with the help of this manager.

3 Class Interfaces

3.1 Client Side

3.1.1 User

Class:	**ProfileScreen**
Profile screen for users where users can see and edit their profile information.	
Attributes:	
name: String	User's name
surname: String	User's surname65
birthDate: Date	User's birthdate
phoneNumber: String	User's phone number
gender: String	User's gender
preferredGender: String	User's preferred gender
photos: Image[]	User's profile photos
hobbies: String[]	User's hobbies
Methods:	
changeInformation(User newUserInformation): boolean	Changes any part of the user's profile information.
addPhoto(Image newPhoto**):** boolean	Adds a new given photo to the user.
deletePhoto(int deletePhotoIndex**):** boolean	Deletes one of the existing photos in the given index.

Table 2: ProfileScreen Class Interface

Class:	**SignUpScreen**
Sign-up screen for users to sign up for the app.	
Attributes:	
name: String	User's name
surname: String	User's surname
birthDate: Date	User's birthdate
phoneNumber: String	User's phone number
password: String	User's password
gender: String	User's gender
preferredGender: String	User's preferred gender
Methods:	
signup(String name, String surname, Date birthDate, String phoneNumber, String password, String gender, String preferredGender): boolean	Sends the information provided by the user to the backend and creates a user account.
showErrors(): JSX.Element	Displays an error message to the user.

Table 3: SignUpScreen Class Interface

Class:	**LoginScreen**
Login screen for users to login to the app.	
Attributes:	
phoneNumber: String	User's phone number
password: String	User's password
Methods:	
login(String phoneNumber,String password)	Sends the login information to the backend. If the login request is valid, the user is directed to the HomeScreen.

Table 4: LoginScreen Class Interface

Class:	**HomeScreen**
Home screen for users to scan their QR codes to attend events.	
Methods:	
openQRScanner(): int	Opens the camera for users to scan the QR code.
joinEvent(int eventID, String phoneNumber): boolean	Adds the user to the event pool so that the user can start matching with others.

Table 5: HomeScreen Class Interface

Class:	**UpcomingEventsScreen**
Upcoming events screen for users to get information about the upcoming events.	
Attributes:	
events: Event[]	List of all upcoming events related to the user
Methods:	
filterEvents(Date date?, Location location?, EventType eventType?, Tag tag): Event[]	Allows users to filter events according to their preferences.
openEventDetails(int eventID): Event	Allows users to see event details such as event description and event type.
searchByName(String name): EventInfo[]	Allows users to search for specific events.

Table 6: UpcomingEventsScreen Class Interface

Class:	**EventHomeScreen**
Event home screen for users to see their matching activity in an event.	
Attributes:	
eventInfo: Event	Event information
likedUsers: int	Number of liked users
match: User	The person that is matched with the current user.
Methods:	
leaveEvent(int eventID): void	Removes the user from the event pool.
startMatching(): User[]	Returns a list of users based on the matching algorithm.
seeCurrentMatch(int userID)	Allows the user to see the current match.
rateEvent(int eventID, int rating, String comment): void	Allows the user to rate and comment about the event.

Table 7: EventHomeScreen Class Interface

Class:	**MessagesScreen**
Message screen for users to message with their matches.	
Attributes:	
chatRecords: ChatRecord[]	Chat records of the user
Methods:	
openChatBox(int userID): ChatRecord	Opens the chatbox with the specified user.
sendMessage(int userID, String message): void	Allows the user to send a message.
terminateChat(int userID): void	Allows the user to terminate the chat.
startChatWithMatch(int userID): ChatRecord	Allows the user to start a chat with a match.

Table 8: MessagesScreen Class Interface

Class:	**MatchingScreen**
Matching screen for users to see other users in the event.	
Attributes:	
abbreviatedName: String	User's abbreviated name, for example: "Rüzgar Ayan" to "Rüzgar A."
age: int	User's age
photos: Image[]	User's profile photos
hobbies: String[]	User's hobbies
commonEventTypes: EventType[]	User's and match's common preferred event types.
matchingScore: int	Measure of users' likeness to each other.
Methods:	
like(int userID): boolean	Allows the user to like another user. Returns true if there is a match and directs the user to FoundMatchScreen.
skip(int userID): void	Allows the user to continue matching.

Table 9: MatchingScreen Class Interface

Class:	**FoundMatchScreen**
Match screen for users to see their current match in the event.	
Attributes:	
abbreviatedName: String	User's abbreviated name, for example: "Rüzgar Ayan" to "Rüzgar A."
age: int	User's age
photos: Image[]	User's profile photos
hobbies: String[]	User's hobbies
commonEventTypes: EventType[]	User's and match's common preferred event types.
matchingScore: int	Measure of users' likeness to each other.
meetingLocation: Location	Meeting location of users
Methods:	
chooseMeetingLocation(Location location): void	Allows the users to choose a meeting location in the event place.
acceptMatch(): void	Directs the user to EventHomeScreen.
removeMatch(): void	Directs the user to MatchingScreen.

Table 10: MatchingScreen Class Interface

Class:	**UIController**
Controls navigation between screens.	
Attributes:	
currentScreen: String	Current screen
loginInformation: User	Current user's information
Methods:	
changeCurrentScreen(String screenName, User loginInformation): JSX.Element	Navigates to the specified screen and changes the display.

Table 11: UIController Class Interface

3.1.2 Event Organizer

Class:	**LoginScreen**
Login screen for event organizers to login to the app.	
Attributes:	
phoneNumber: String	Event organizer's phone number
password: String	Event organizer's password
Methods:	
login(String phoneNumber, String password)	Sends the login information to the backend. If the login request is valid, the user is directed to OrganizedEventsScreen.

Table 12: LoginScreen Class Interface

Class:	**OrganizedEventsScreen**
The main screen where the event organizers see all their past and future events.	
Attributes:	
organizedEvents: Event[]	Full information about all the events organized by the event organizer
Methods:	
cancelEvent(int eventID): void	Allows event organizers to cancel events.
updateEventDetails(int eventID, Event updatedEvent**):** void	Allows event organizers to update event details with newly provided information.
viewEventDetails(int eventID**):** Event	Allows event organizers to view event details.
viewEventStatistics(int eventID**):** EventStatistics	Allows event organizers to view event statistics such as number of users in the event pool and number of matches.
promoteEvent(int eventID): void	Allows event organizers to advertise their events so that their events can be displayed at the top of the event list.

Table 13: OrganizedEventsScreen Class Interface

Class:	**EventCreationScreen**
Event creation screen where event organizers can schedule a new event.	
Attributes:	
eventName: String	Event name
startTime: Date	Start time of the event
endTime: Date	End time of the event
eventPlace: Location	Location where the event takes place
chosenPartition: Partition	The chosen partition of the event place with different subareas to meet
tag: Tag[]	Tags to categorize the event
eventPhoto: String	Event photo
Methods:	
submitForConfirmation(String eventName, Date startDate, Date endDate, Location eventPlace, Partition chosenPartition, Tag[] tag, String eventPhoto): void	Allows event organizers to create events with the specified information.

Table 14: EventCreationScreen Class Interface

Class:	**EventLocationsScreen**
Event location screen where event organizers can add new event locations and new event location partitions.	
Attributes:	
eventLocations: Location[]	All the event locations added by all the event organizers.
Methods:	
addNewEventLocation(Location location): void	Allows event organizers to add new event locations.
addEventLocationPartition(Partition partition): void	Allows event organizers to add new event location partitions where users can meet after they match.
viewEventsAtLocation(Location location): Event[]	Allows event organizers to search for events at specific locations.

Table 15: EventLocationsScreen Class Interface

Class:	**UIController**
Controls navigation between screens.	
Attributes:	
currentScreen: String	Current screen
loginInformation: EventOrganizer	Current event organizer's information
Methods:	
changeCurrentScreen(String screenName, EventOrganizer loginInformation): JSX.Element	Navigates to the specified screen and changes the display.

Table 16: UIController Class Interface

3.2 Server Side

Class:	**User**
User is Django's base User model	
Attributes	
username: String	
email: String	
password: String	

Table 17: User Class Interface

Class:	**Profile**
Profile has a one to one relation with User	
Attributes	
name: String	
surname: String	
phone_number: String	
date_of_birth: Date	
description: String	
gender: String	
preference: String	
profile_picture: String	
Methods	
updateProfile(**profile_data): bool	Updates the profile with the new data
updateProfilePicture(string picture_link): bool	Updates the profile picture
changePassword(string oldPassword, string newPassword): bool	Updates the old password with the new password

Table 18: Profile Class Interface

Class:	**Event**
Event allows user to match with one another	
Attributes	
event_name: String	
event_description: String	
event_tags: Tag[]	
event_start_time: Date	
event_finish_time: Date	
event_creator: User	
event_location: Location	
event_attendees: User	
Methods	
updateEventDetails(**details): bool	
updateTags(**tags): bool	

Table 19: Event Class Interface

Class:	**Organizer**
Extends User class with additional attributes for event organizers.	
Attributes	
user: User	Inherits attributes from User class.
organizer_phone: String	Phone number of event organizer.
organizer_contact: String	Additional contact info of the event organizer.
organizer_rating: double	Rating of the event organizer.
Methods	
updateRating(int rating): double	Calculates the new rating using input rating and the old rating. Updates the organizer's rating with the calculated rating afterwards.
updateOrganizerPhone(String phone_number): Organizer	Updates the organizer_phone attribute.
updateOrganizerContact(String contact): Organizer	Updates the organizer_contact attribute.

Table 20: Organizer Class Interface

Class:	**Comment**
Comment class is used to leave comment to events	
Attributes	
commenter: User	The user that sends the comment.
event: Event	The event which is commented on.
rating: int	The overall score given to the event.
comment_text: String	The comment itself.

Table 21: Comment Class Interface

Class:	**Match**
Match class is used to represent two matching users in an event.	
Attributes	
user1: User	First user of the match.
user2: User	Second user of the match.
event: Event	The event where the match happened.

Table 22: Match Class Interface

Class:	**Location**
Location class is used to represent event locations.	
Attributes	
name: String	Name of the location.
description: String	Description of the location.
address: String	Address of the location.
Methods	
updateName(String location_name): Location	Updates the name attribute.
updateDescription(String location_description): Location	Updates the description attribute.
updateLocation(String location_address): Location	Updates the address attribute.

Table 23: Location Class Interface

Class:	**Ticket**
Ticket class is used for handling user reports.	
Attributes	
reporter: User	The user that creates the ticket.
reported: User	The user that's reported.
description: String	Description of the ticket.

Table 24: Ticket Class Interface

Class:	**Tag**
Tag class is used for handling event tags.	
Attributes	
tag_name: String	Tag's name.
tag_description: String	Tag's description.

Table 25: Tag Class Interface

Admin

Admin class is not represented as a separate model even though the functionality differs from a normal user. Django REST Framework handles the additional admin privileges itself without the need of an additional model.

References

[1] *Object-Oriented Software Engineering, Using UML, Patterns, and Java, 2nd Edition*, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2004, ISBN: 0-13-047110-0.