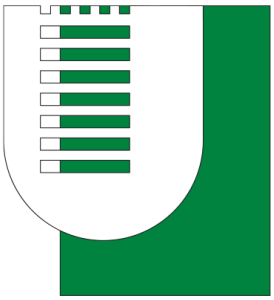


TOR VERGATA

UNIVERSITÀ DEGLI STUDI DI ROMA

Corso di Laurea in Ingegneria Informatica

Ingegneria del Software e Progettazione Web

A.A. 2023/2024

Technical Documentation

NightPlan

Musical Events and Shows Management Software

Professors

Prof. Davide Falessi
Prof. Guglielmo De Angelis

Students

Nicolas Oberi
Matteo Trossi

1 Software Requirements Specification

1.1 Introduction

1.1.1 Aim of the document

The aim of this document is to outline the functional and non-functional specifications for the NightPlan software. It provides a comprehensive vision of the software's features and highlights any discrepancies between the final project and the initial prototype. This document serves as a crucial reference for developers, testers, and project stakeholders, ensuring a clear understanding of the project's scope and requirements.

Additionally, it details the fundamental hardware and software requirements necessary for NightPlan's operation. Included in this document is a link to our SonarCloud dashboard, which grants access to our GitHub repository. From this dashboard, you can review the analysis of bugs, code smells, and security vulnerabilities within the software, as well as the steps taken to address these issues throughout the development process.

The source code, along with all other deliverables, is available in the attached .zip folder. This comprehensive package provides everything needed for a total understanding and evaluation of the NightPlan project.

1.1.2 Overview of the defined system

NightPlan is a standalone desktop application designed to assist users in discovering local events and making new friends. It provides a unified platform where users can browse all organized events in their city. Additionally, NightPlan offers event organizers a powerful tool to reach a broad audience without the need to promote their events through other channels such as social media, websites, or television.

This centralized approach makes it easier for both participants and organizers, improving the overall experience and community engagement.

1.1.3 Hardware and software requirements

Hardware Requirements

To ensure optimal performance of the NightPlan application, the following hardware specifications are recommended:

- **Processor:** Intel Core i3 or equivalent AMD
- **Memory (RAM):** 2 GB minimum
- **Display:** 1280 x 800 resolution or higher

Software Requirements

The NightPlan application requires the following software components:

- **Operating System:** Windows 10 or higher, macOS 10.14 or higher, Linux (kernel 3.10 or higher)
- **Java Runtime Environment (JRE):** JRE 17 or higher

These hardware and software requirements ensure that NightPlan runs efficiently and provides a good user experience.

1.1.4 Related systems, pros and cons

Meetup: This platform enables users to organize and participate in events based on their interests. Users can join specific groups and attend events organized by these groups. However, Meetup restricts participation to group-specific events. In contrast, NightPlan allows users to access and participate in all events organized in their city, providing a more comprehensive event discovery experience.

Eventbrite: This event management platform allows organizers to create and promote events, while users can discover and participate in both local and online events. However, Eventbrite does not support the creation of groups for social interaction. NightPlan, on the other hand, allows users to create and join groups where they can chat and get to know each other, creating a sense of community and making event participation more engaging.

1.2 User Stories

1.2.1 Nicolas Oberi

1. As a user, I want to be able to see all the events in my city, so that I don't have to search from multiple sources*.
2. As a user, I want to be able to book my attendance for an event, so that I am sure I can attend the event.
3. As an organizer, I want people to know when I post a new event in their city, so that I can have more customers.

*sources: social media pages, webpages

1.2.2 Matteo Trossi

1. As a user, I want to be able to chat with other users participating to the same event, to make new friends.
2. As an organizer, I want to be able to see analytics* about my past events, so that I can improve the quality and organization of future events.
3. As an organizer, I want to be able to edit an event, so that I can keep users informed about changes.

*analytics: times event was clicked by users, planned participants, actual participants

1.3 Functional requirements

1.3.1 Nicolas Oberi

1. The system shall display available events in user's city.
2. The system shall show photos and information provided by the organizer in the place's page.
3. The system shall provide a past events archive for the organizer.

1.3.2 Matteo Trossi

1. The system shall provide a group chat with users participating to the same event.
2. The system shall provide an interface to add, update or delete events created by the organizer.

3. The system shall provide an analytics page for the event organizer.

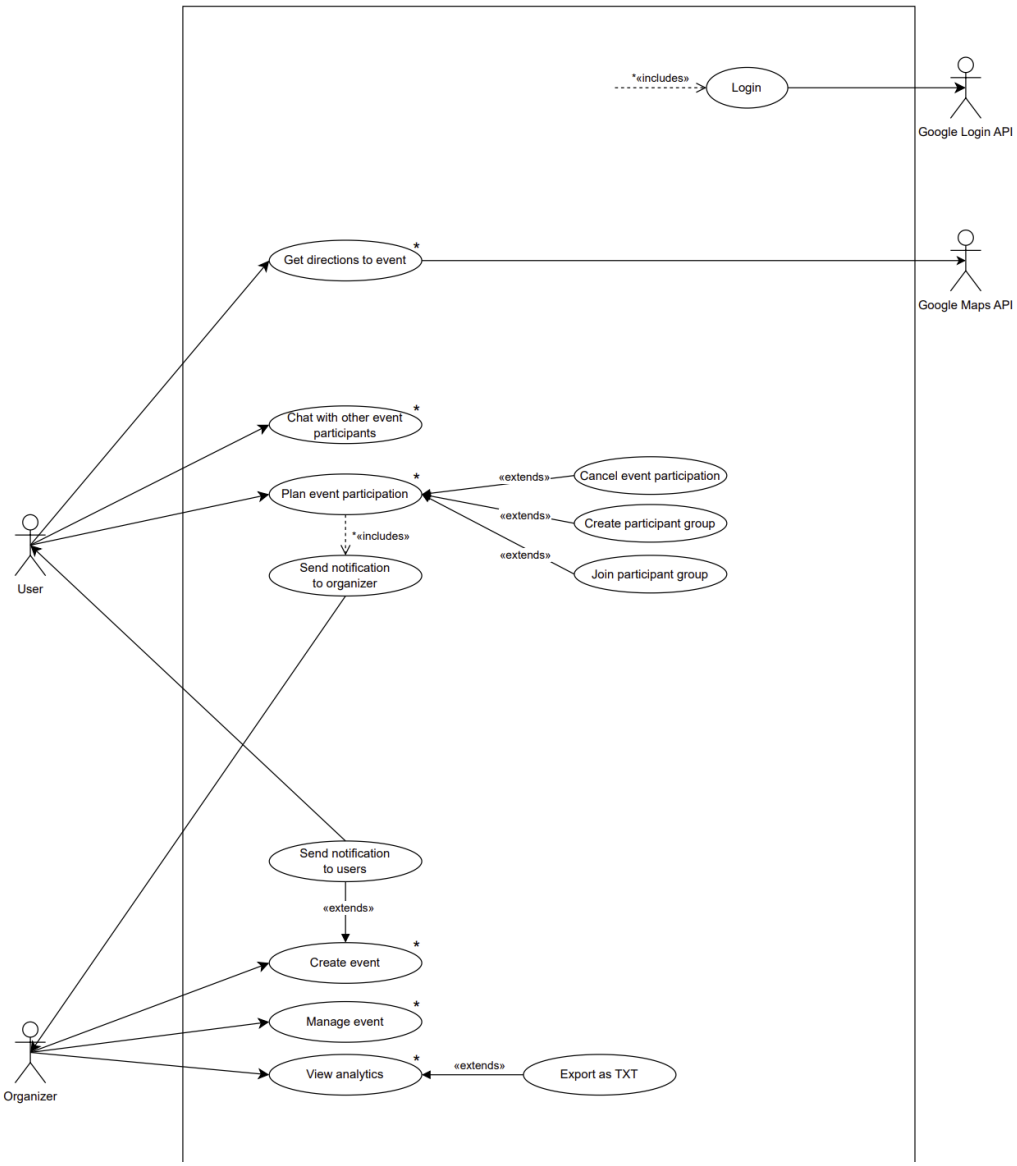
1.4 Use cases

Division of Work on Deliverables

In our project, the two group members divided the work based on different use cases. Nicolas Oberi and Matteo Trossi each took responsibility for specific aspects of the project:

- **Nicolas Oberi** has worked on the "Plan Event Participation" use case.
- **Matteo Trossi** has worked on the "Create Event" use case.

1 Overview diagram



2 Internal steps

Nicolas Oberi

Use case: *Plan event participation*

1. The system retrieves all the available events in the user's city from the system catalogue.

2. The user selects an event.

3. The system shows the user the information about the event.

4. The system asks the user if he wants to participate.*

5. The user accepts to participate.

6. The system notifies the user his participation is confirmed.

7. The system asks the user if he wants to create (or join) an event group.*

8. The system notifies the event organizer of a new participation to his event.

Extensions:

1a: System catalogue doesn't respond: notify the user and end the use case.

1b: No events available in user's city: notify the user and end the use case.

5a: The user doesn't accept to participate: go back to the home page and end the use case.

* := discrepancies

Matteo Trossi

Use case: *Create event*

1. The system verifies if the user is logged as an organizer.

2. The organizer selects "Add a new event".

3. The system prepares a blank creation form.

4. The organizer adds all the information in the blank fields.

5. The system uses Google Maps API to verify the correctness of the location address. *

6. The organizer completes the creation of the event.

7. The system connects to the internal database and updates it with new event data.

8. The system shows a confirmation popup organizer of the successful creation of the event.

9. The system notifies all users in event city that a new event is available.

Extensions:

4a. Input text is too long: The system asks the organizer to insert a shorter value in the specific text field.

4b. Invalid time format: The system asks the organizer to insert a valid time.

4c. Selected date is before the current date or empty: The system asks the organizer for a valid date.

5a. Google Maps API nonresponding: The system notifies the organizer that the API is not responding and goes back to step 3. *

6a. Event with the same name already created: System notifies the organizer to change name.

7a. Database failure: The system notifies the organizer that the creation failed and terminates the use case.

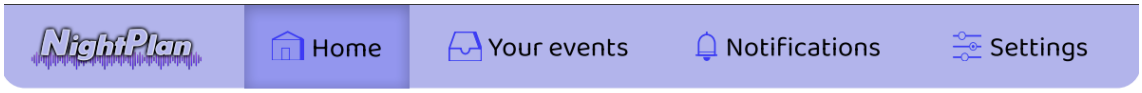
Dictionary:

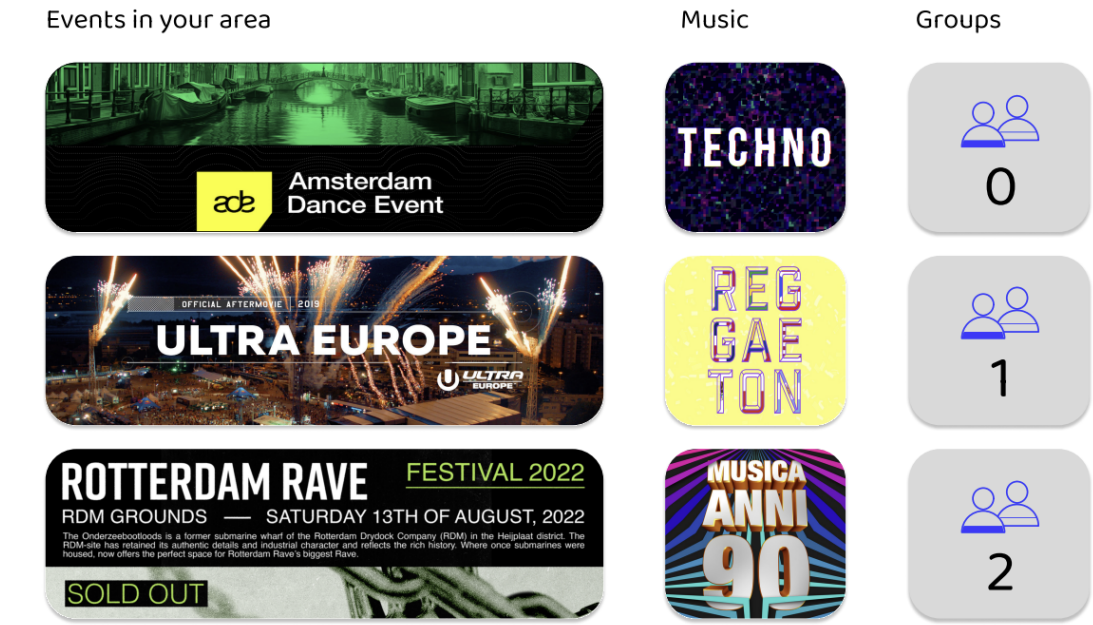
1. Information: event name, address, province, city, date from the calendar, time, type of music from a preloaded list, an image picked from user files.

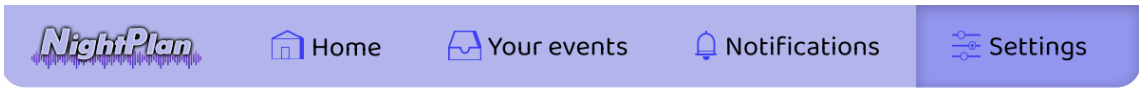
* := discrepancies

2 Storyboards

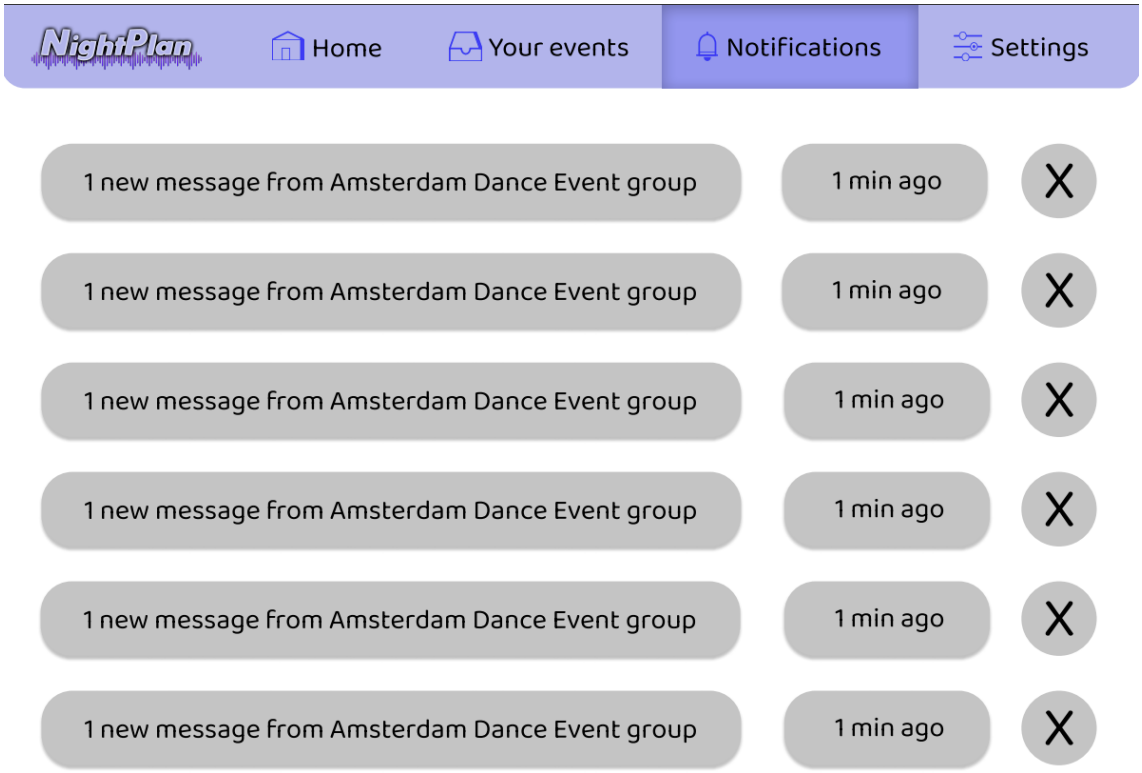
We designed three storyboards for each member of the work group using Figma online design tool.


NightPlan
Home
Your events
Notifications
Settings

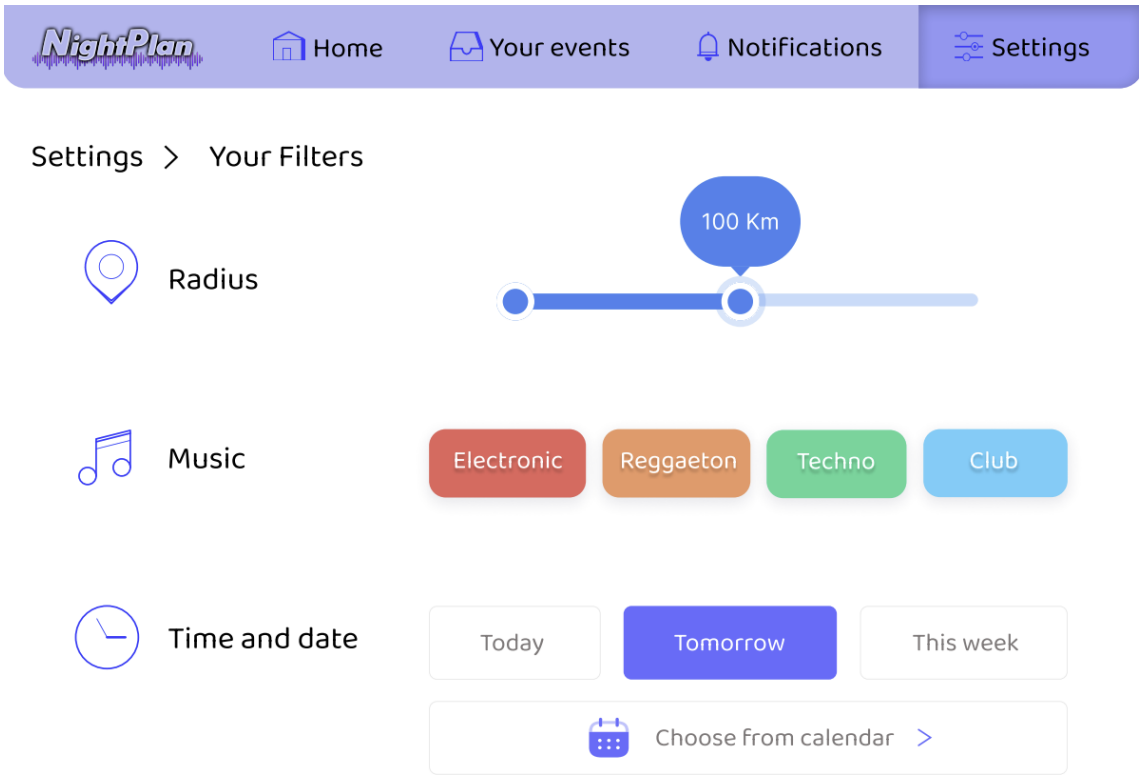

Events in your area
Music
Groups
Amsterdam Dance Event
ade
TECHNO
0
OFFICIAL AFTERMOVIE | 2019
ULTRA EUROPE
REG GAE TON
1
ROTTERDAM RAVE
FESTIVAL 2022
RDM GROUNDS — SATURDAY 13TH OF AUGUST, 2022
The Onderzeebootloods is a former submarine wharf of the Rotterdam Drydock Company (RDM) in the Heijplaat district. The RDM-site has retained its authentic details and industrial character and reflects the rich history. Where once submarines were housed, now offers the perfect space for Rotterdam Rave's biggest Rave.
SOLD OUT
MUSICA ANNI 90
2

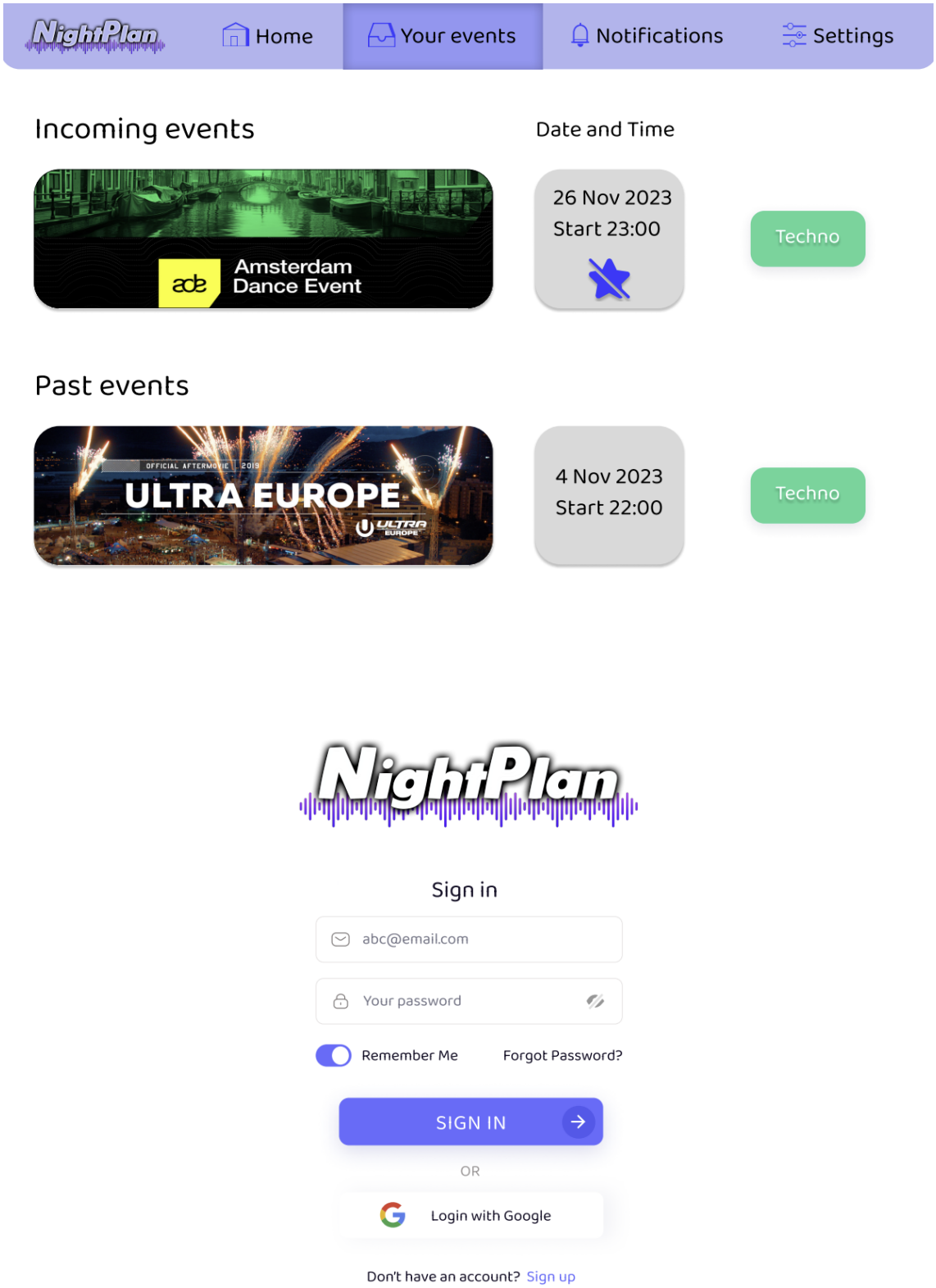

NightPlan
Home
Your events
Notifications
Settings


Your profile
Your groups
Your filters
Contact us
Help & FAQs
Sign out



Matteo Trossi




NightPlan
Home
Your events
Notifications
Settings
Incoming events
Date and Time
Amsterdam Dance Event
26 Nov 2023
Start 23:00
Techno
Past events
OFFICIAL AFTERMOVIE 2019
ULTRA EUROPE
4 Nov 2023
Start 22:00
Techno
NightPlan
Sign in
abc@email.com
Your password
Remember Me
Forgot Password?
SIGN IN
OR
Login with Google
Don't have an account? Sign up

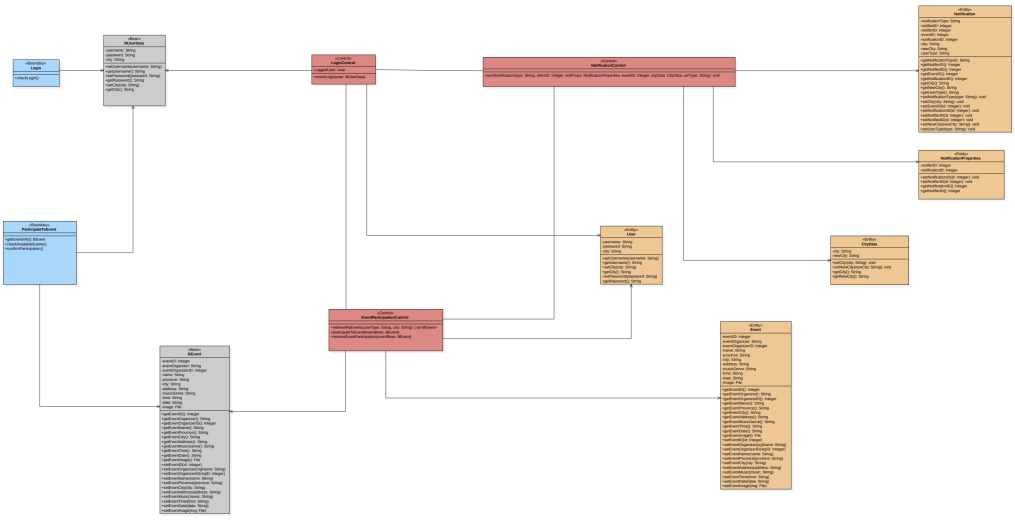
3 Design

3.1 Class Diagram

3.1.1 VOPC (View of participating classes)

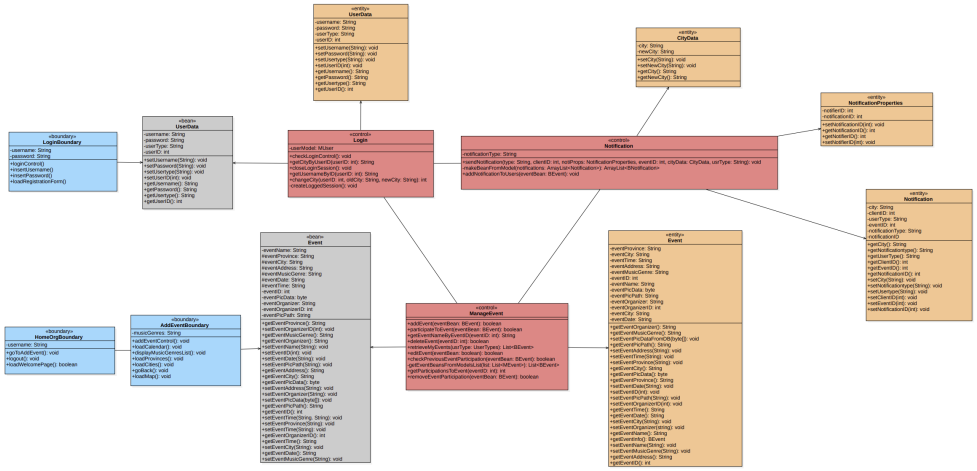
Nicolas Oberi

Use Case: *Plan event participation*



Matteo Trossi

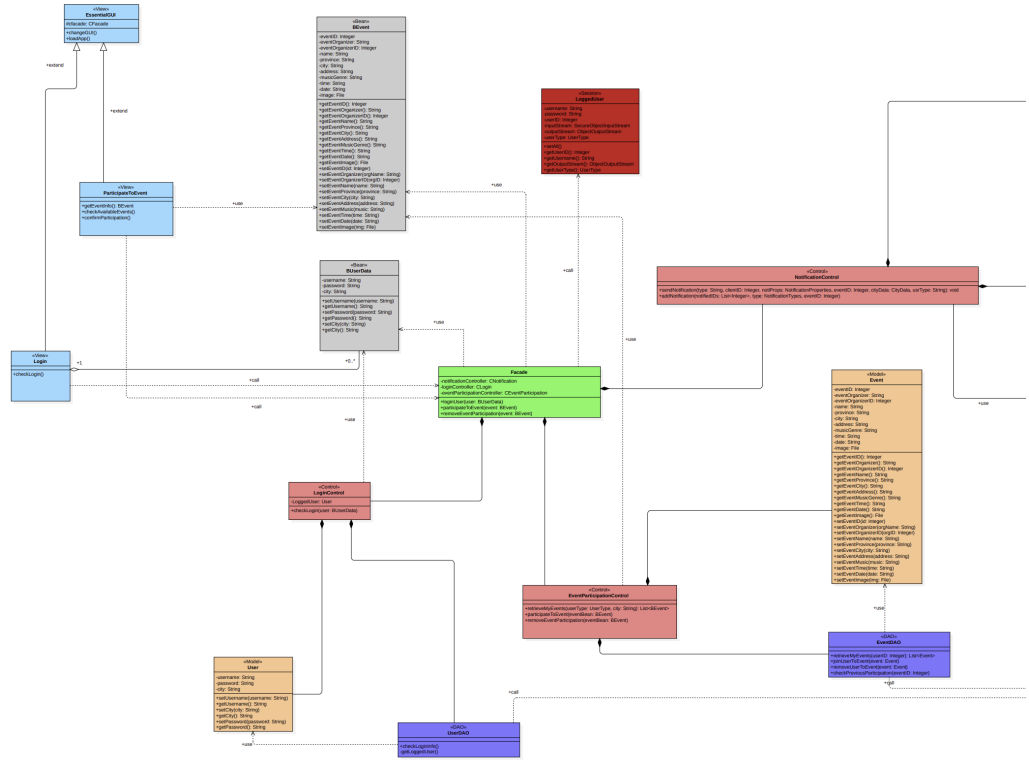
Use Case: *Create event*

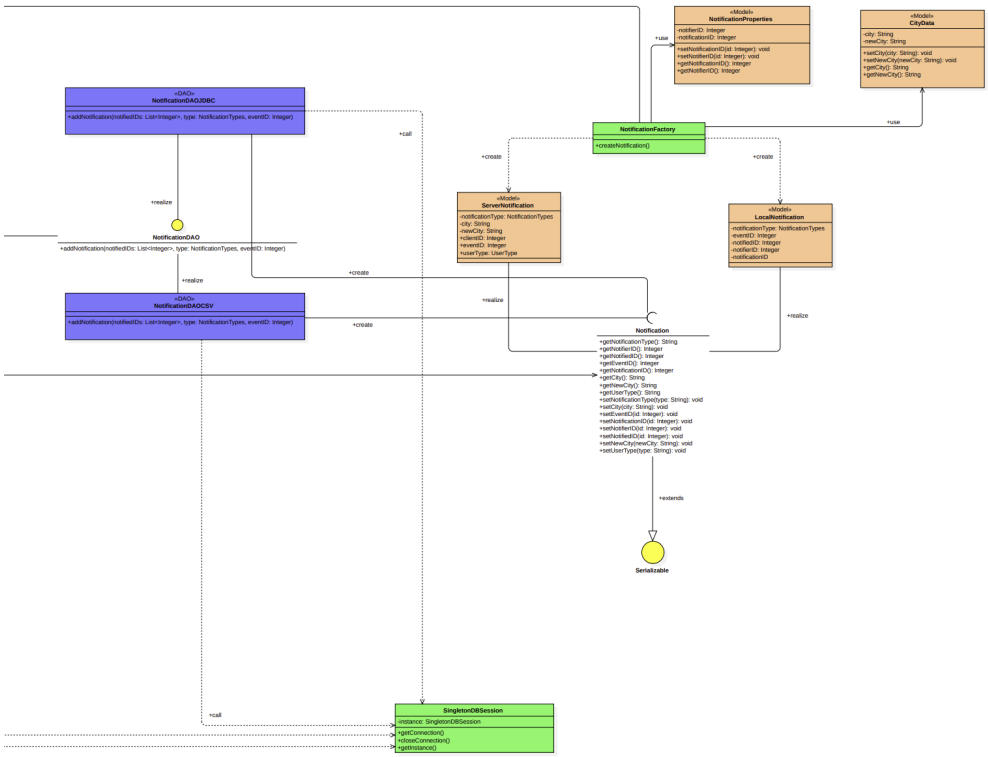


3.1.2 Design-Level Diagram

Nicolas Oberi

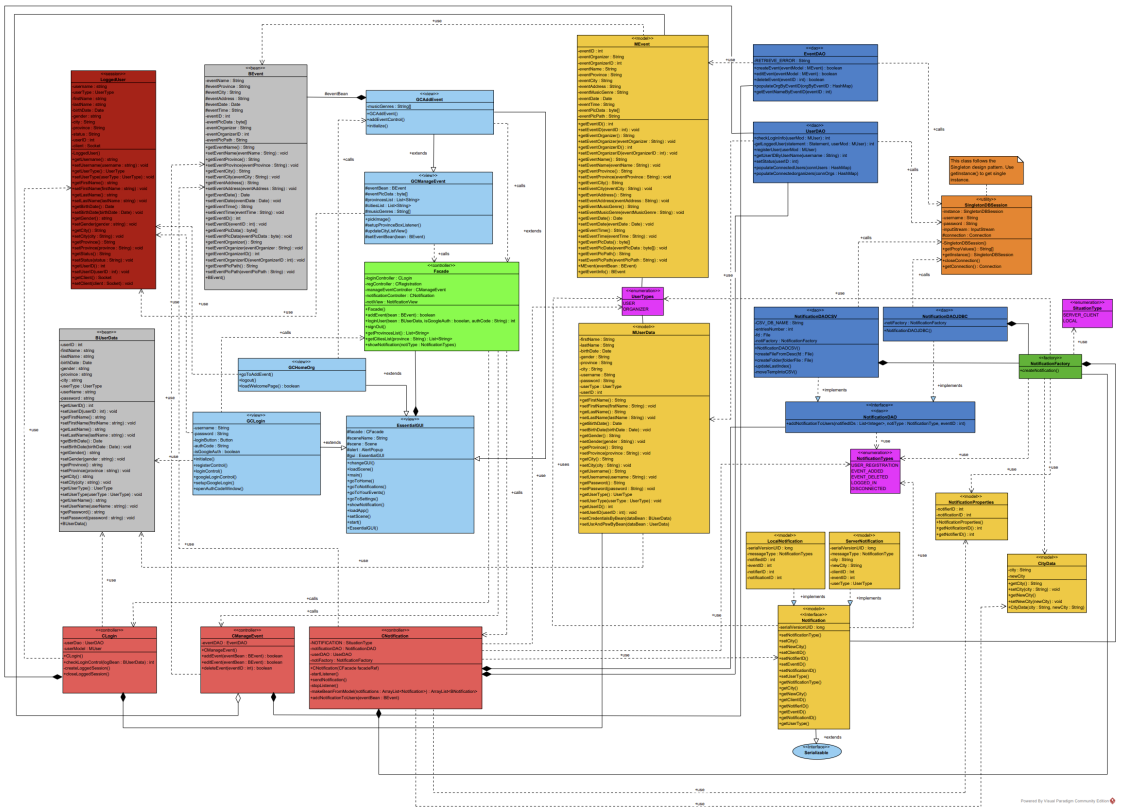
Use Case: *Plan event participation*


«View»
EssentialGUI
#cfacade: CFacade
+changeGUI()
+loadMap()
«Bean»
BEvent
-eventID: Integer
-eventOrganizer: String
-eventOrganizerID: Integer
-name: String
-province: String
-city: String
-address: String
-musicGenre: String
-time: String
-date: String
-image: File
+getEventID(): Integer
+getEventOrganizer(): String
+getEventOrganizerID(): Integer
+getEventName(): String
+getEventProvince(): String
+getEventCity(): String
+getEventAddress(): String
+getEventMusicGenre(): String
+getEventTime(): String
+getEventDate(): String
+getEventImage(): File
+setEventID(id: Integer)
+setEventOrganizerName(name: String)
+setEventOrganizerID(orgID: Integer)
+setEventNameName(name: String)
+setEventProvince(province: String)
+setEventCity(city: String)
+setEventAddress(address: String)
+setEventMusic(music: String)
+setEventTime(time: String)
+setEventDate(date: String)
+setEventImage(img: File)
«Validator»
LoggedUser
-username: String
-password: String
-userID: Integer
-inputStream: SecureObjectInputStream
-outputStream: ClipboardObjectStream
-userType: UserType
+getID()
+getUserID(): Integer
+getUsername(): String
+getOutputStream(): ClipboardObjectStream
+getUserType(): UserType
«View»
ParticipateToEvent
+getEventInfo(): BEvent
+checkAvailableEvents()
+confirmParticipation()
«Bean»
BUserData
-username: String
-password: String
-city: String
+setUsername(username: String)
+getUsername(): String
+setPassword(password: String)
+getPassword(): String
+setCity(city: String)
+getCity(): String
«Controll»
NotificationControl
+sendNotification(type: String, clientID: Integer, notifProps: NotificationProperties, eventID: Integer, cityData: CityData, userType: String): void
+addNotification(notifiedIDs: List<Integer>, type: NotificationTypes, eventID: Integer)
«View»
Login
+checkLogin()
Facade
-notificationController: CNotification
-loginController: CLogin
-eventParticipationController: CEventParticipation
+loginUser(user: BUserData)
+participateToEvent(event: BEvent)
+removeEventParticipation(event: BEvent)
«Model»
Event
-eventID: Integer
-eventOrganizer: String
-eventOrganizerID: Integer
-name: String
-province: String
-city: String
-address: String
-musicGenre: String
-time: String
-date: String
-image: File
+getEventID(): Integer
+getEventOrganizer(): String
+getEventOrganizerID(): Integer
+getEventName(): String
+getEventProvince(): String
+getEventCity(): String
+getEventAddress(): String
+getEventMusicGenre(): String
+getEventTime(): String
+getEventDate(): String
+getEventImage(): File
+setEventID(id: Integer)
+setEventOrganizerByName(name: String)
+setEventOrganizerID(orgID: Integer)
+setEventNameName(name: String)
+setEventProvince(province: String)
+setEventCity(city: String)
+setEventAddress(address: String)
+setEventMusic(music: String)
+setEventTime(time: String)
+setEventDate(date: String)
+setEventImage(img: File)
«Control»
LoginControl
+LoggedUser: User
+checkLogin(user: BUserData)
«Control»
EventParticipationControl
+retrieveMyEvents(userType: UserType, city: String): List<BEvent>
+participateToEvent(event: BEvent)
+removeEventParticipation(event: BEvent)
«Model»
User
-username: String
-password: String
-city: String
+setUsername(username: String)
+getUsername(): String
+setCity(city: String)
+getCity(): String
+setPassword(password: String)
+getPassword(): String
«DAO»
EventDAO
+retrieveAllEvents(eventID: Integer): List<Event>
+getUserToEvent(event: Event)
+removeUserToEvent(event: Event)
+checkPreviousParticipation(eventID: Integer)
«DAO»
UserDAO
+checkLogin(info)
+getLoggedUser()


«Model»
NotificationProperties
-notifierID: Integer
-notificationID: Integer
+setNotificationID(id: Integer): void
+setNotifierID(id: Integer): void
+getNotificationID(): Integer
+getNotifierID(): Integer
«Model»
CityData
-city: String
-newCity: String
+setCity(city: String): void
+setNewCity(newCity: String): void
+getCity(): String
+getNewCity(): String
«DAO»
NotificationDAOJDBC
+add(Notification(notifiedIDs: List<Integer>, type: NotificationTypes, eventID: Integer)
NotificationFactory
+createNotification()
«Model»
ServerNotification
-notificationType: NotificationTypes
-city: String
-newCity: String
-clientID: Integer
-eventID: Integer
-userType: UserType
«Model»
LocalNotification
-notificationType: NotificationTypes
-eventID: Integer
-notifiedID: Integer
-notifierID: Integer
-notificationID
NotificationDAO
+addNotification(notifiedIDs: List<Integer>, type: NotificationTypes, eventID: Integer)
«DAO»
NotificationDAOCSV
+add(Notification(notifiedIDs: List<Integer>, type: NotificationTypes, eventID: Integer)
Notification
+getNotificationType(): String
+getNotifierID(): Integer
+getNotifiedID(): Integer
+getEventID(): Integer
+getNotificationID(): Integer
+getCity(): String
+getNewCity(): String
+getUserType(): String
+setNotificationType(type: String): void
+setCity(city: String): void
+setEventID(id: Integer): void
+setNotificationID(id: Integer): void
+setNotifierID(id: Integer): void
+setNotifiedID(id: Integer): void
+setNewCity(newCity: String): void
+setUserType(type: String): void
Serializable
SingletonDBSession
-instance: SingletonDBSession
+getConnection()
+closeConnection()
+getInstance()

Matteo Trossi

Use Case: *Create event*



3.2 Design Patterns

Within the NightPlan application, we employed various design patterns to enhance code structure and simplify specific tasks. Matteo Trossi took charge of the Factory pattern, while Nicolas Oberi focused on the Observer pattern.

Singleton Pattern for Database Connections

We implemented the Singleton pattern to ensure the existence of a single instance of the database connection. This allowed us to avoid inconsistencies and duplications in data access operations performed by DAO classes. With this pattern, all classes requiring database access can do so through a single instance of connection, without needing to pass the connection as a parameter in every constructor.

Observer Pattern for Notification Dispatch

We employed the Observer pattern to manage notification delivery to users. With this pattern, we defined a subject that sends notifications to registered observers. Nicolas Oberi led the implementation of this pattern, ensuring that observers are notified whenever relevant events occur within the application, such as events creation or users participation.

Factory Pattern for Notifications

Matteo Trossi led the implementation of the Factory pattern, which was utilized for creating chat messages and notifications within the NightPlan application. In addition, in the context of the Factory design pattern, we have also added a factory class for observers. This pattern facilitated the creation of message and notification objects, providing a flexible and maintainable approach to object instantiation.

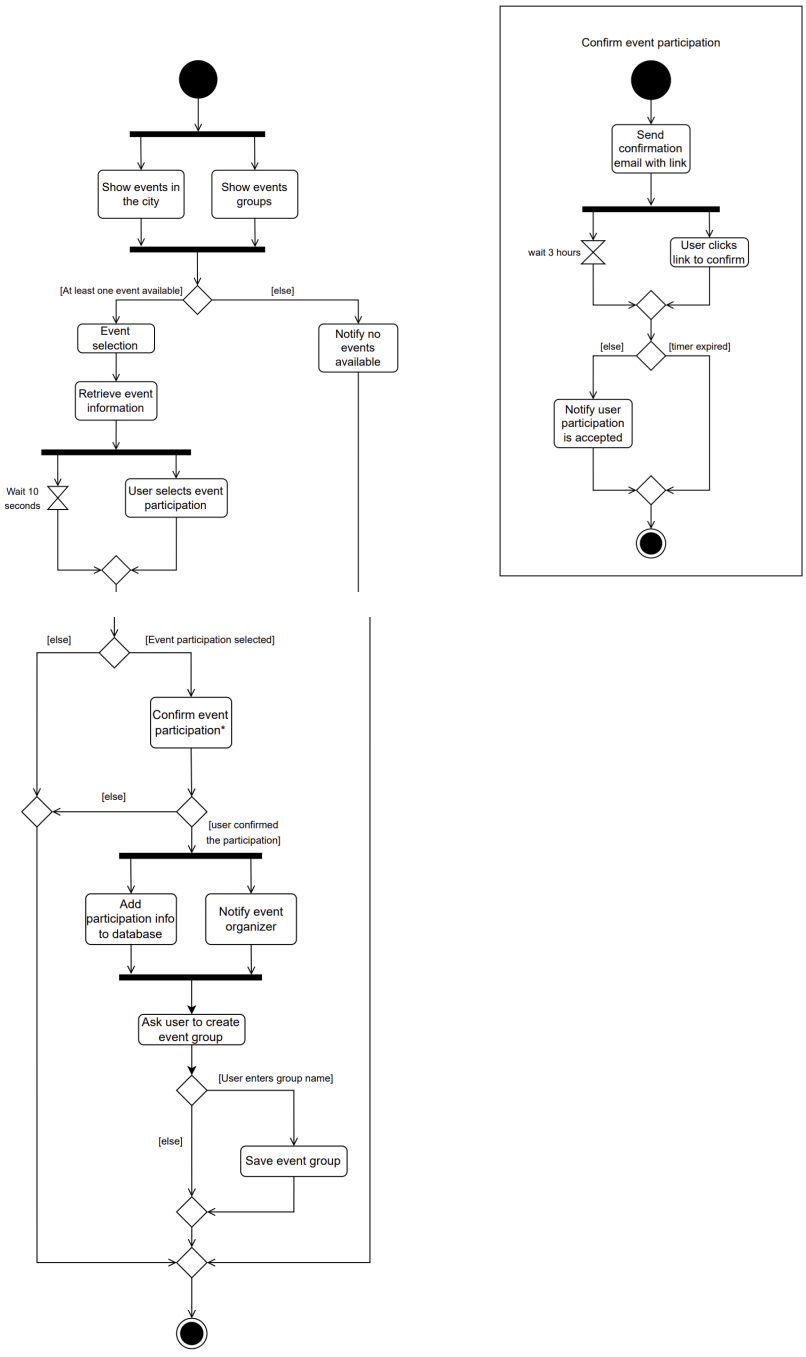
Facade Pattern for Unified Controller Management

Additionally, we applied the Facade pattern for overall management of controller methods exchange between graphical and application controllers. This pattern simplified the interface between different components of the application, enhancing code readability and maintainability.

3.3 Activity Diagram

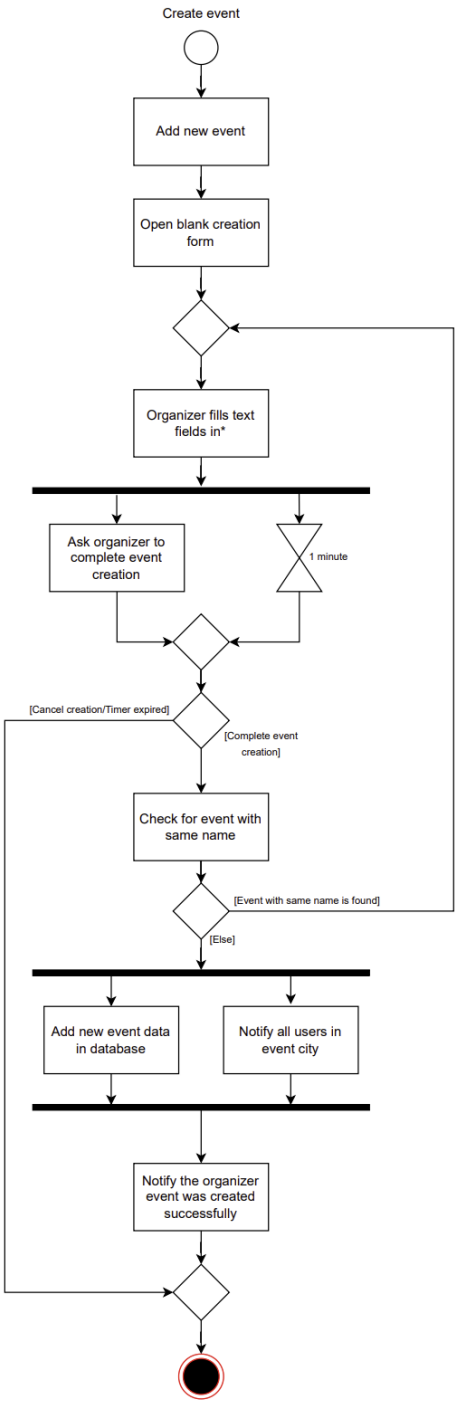
Nicolas Oberi

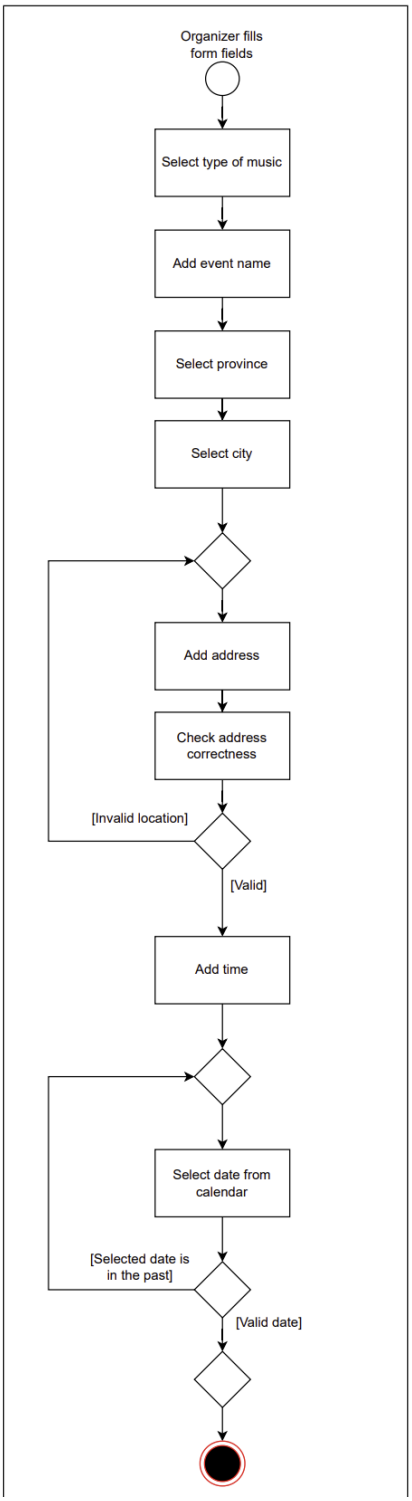
Use Case: *Plan event partecipation*



Matteo Trossi

Use Case: *Create event*

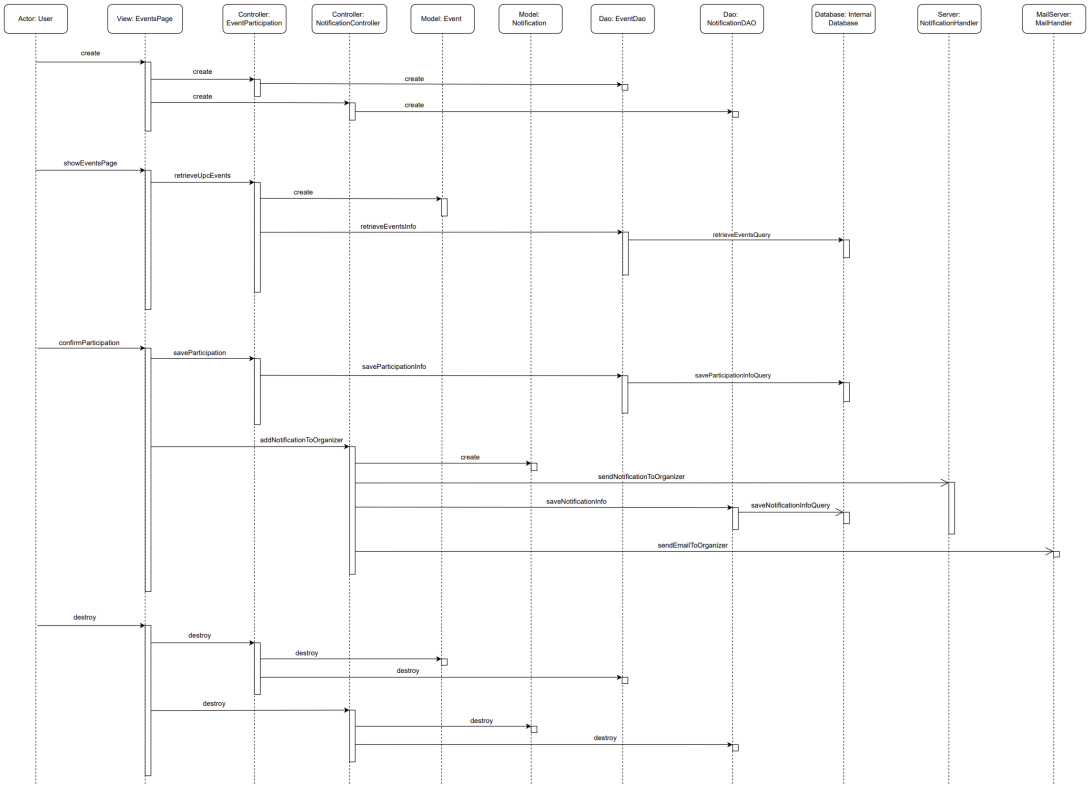




3.4 Sequence Diagram

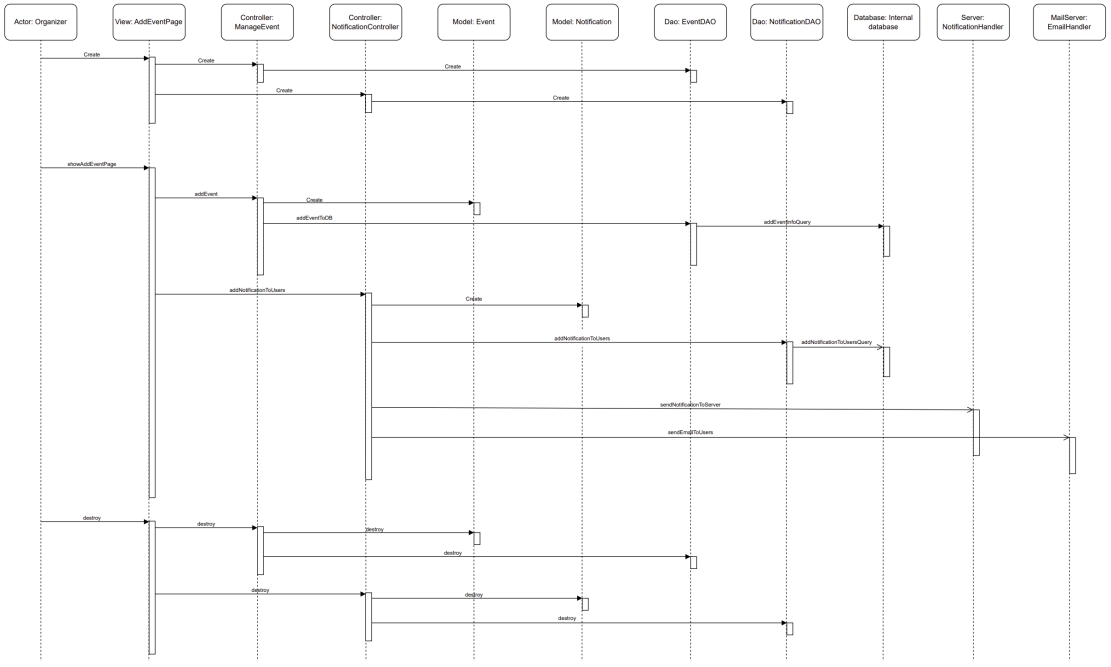
Nicolas Oberi

Use Case: *Plan event participation*



Matteo Trossi

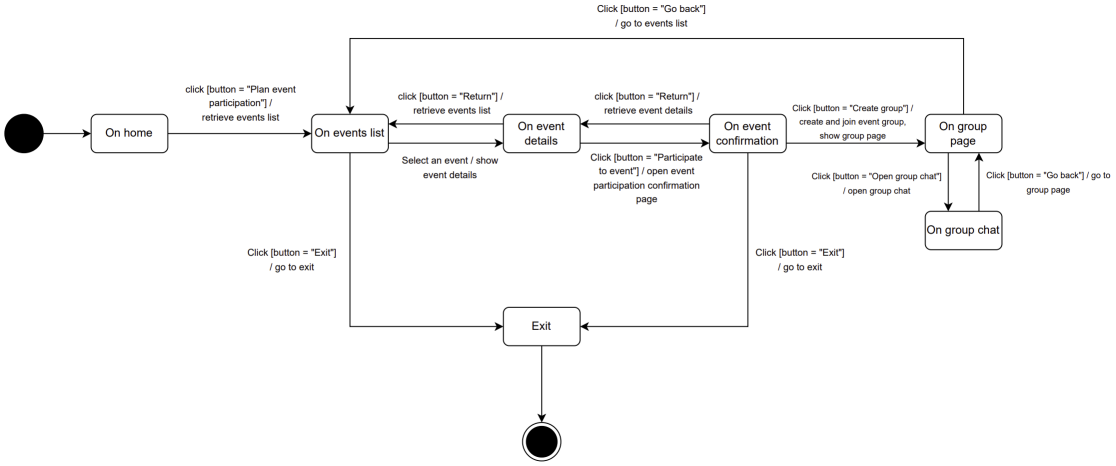
Use Case: *Create event*



3.5 State Diagram

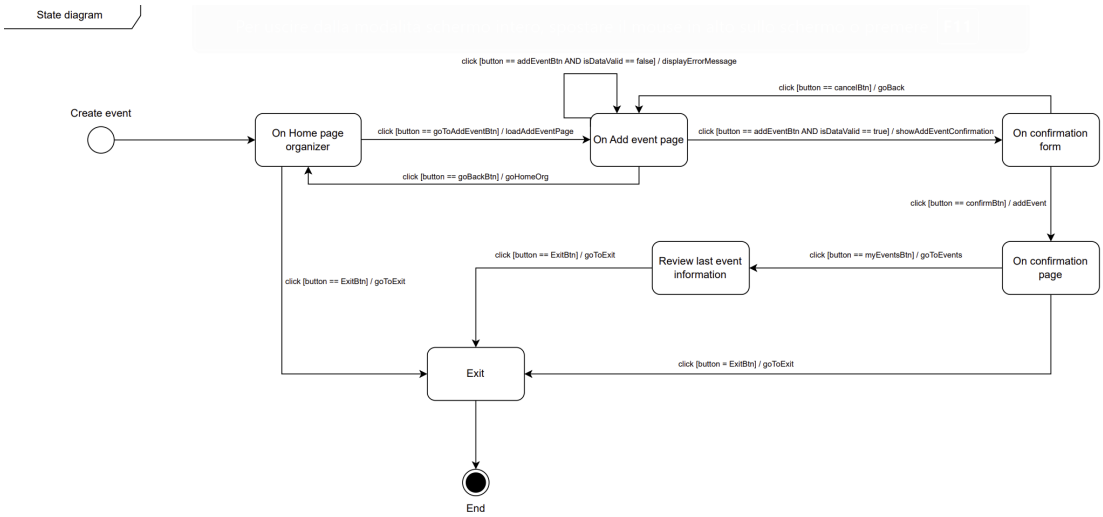
Nicolas Oberi

Use Case: *Plan event participation*



Matteo Trossi

Use Case: *Create event*



3.6 Discrepancies

The following list outlines discrepancies identified between the final version of the application and its prototype.

Nicolas Oberi

1. **UseCaseSteps**: The system doesn't ask the user to participate. They system doesn't ask the user to create or join a group right after participating to the event.

2. **Activity Diagram**: Confirm event participation doesn't happen with confirmation link via email.

3. **State Diagram**: After event confirmation the user isn't immediately asked to create or join the event group.

4. **Sequence Diagram**: Mail Server doesn't exist in final application.

Matteo Trossi

4. **Use Case Steps**: Google Maps non utilized to check address correctness.

5. **Activity Diagram**: Address correctness isn't verified. The form doesn't auto close after 1 minute.

6. **State Diagram**: Confirmation page doesn't exist after confirmation form. After confirmation page there is no event recap, in the application we immediately go to "your events" page.

7. **Storyboards**: Filters and bookmarks missing in user settings.

8. **Sequence Diagram**: Mail Server doesn't exist in final application. sendEmailToUsers() method wasn't implemented in final application.

4 Testing

To ensure the robustness and reliability of our application, we have developed six unit tests for our controllers using JUnit library. We divided the work among team members, with each person responsible for writing tests for three classes. This approach allowed us to effectively cover all key areas of the application. Here's an overview of how we implemented the test classes:

Division of Responsibilities

We divided the test classes between the two members of our group as follows:

- **Nicolas Oberi:**
    - **TestManageEventController**: used to test the creation of new event and the editing of a casual event previously created
    - **TestNotificationController**: used to verify the adding of a new notification
    - **TestChatController**: used to test the functionality of the chat controller, esuring the correct sending of a message
- **Matteo Trossi:**
    - **TestLoginController**: used to verifiy the correctness of a new user registration and the change of city for a registered user
    - **TestGroupController**: used to test creation of a user group and joining of a group
    - **TestEventParticipationController**: used to test different functionalities related to the event participation, such as the participation of an event, the removal of the participation and the verification of a previous participation to the event

5 Exceptions

We have developed various custom exception classes in Java to handle error situations appropriately within our application.

The development of these custom exception classes was driven by the need for:

- **Specificity in Error Handling:** Custom exceptions allow us to handle specific error scenarios more precisely, providing meaningful error messages and responses to each situation.
- **Improved Code Readability:** By defining our own exception classes, we make the code more readable and maintainable. It becomes clearer what types of errors are being handled and how they are addressed.
- **Enhanced Debugging:** With specific exception types, it is easier to debug issues. The stack traces and error logs become more informative, leading in faster resolution of problems.

Below is a brief overview of the custom exception classes we have implemented:

1. **DuplicateEventParticipation**: this exception is thrown when a user have previously joined ad event.
2. **EventAlreadyAdded**: this exception is thrown when an organizer tries to create an event with the same name as another.
3. **EventAlreadyDeleted**: this exception handles the situazione where an event is deleted by the organizer while a user, who previously joined it, is on "Your Events" page and tries to create a new group or join an existing group for the deleted event.
4. **GroupAlreadyCreated**: this exception is thrown when a user tries to create a new group for an event which has already a group. This situation happens when a user creates a group while another user has already completed the group creation process.
5. **InvalidGroupName**: this exception handles errors related to the creation of a group with an empty name tag.
6. **InvalidTokenValue**: this exception is used for error related to login with Google account operations. In particular, it is thrown when a user or an organizer put a wrong/invalid token given by Google to continue the login process.
7. **InvalidValueException**: this exception is thrown when a user inserts an invalid value in a form, such as an invalid date/time format.
8. **MinimumAgeException**: this exception is thrown during the registration process, when a user adds an age below 18 years old.
9. **TextTooLongException**: this exception is user to notificate users or organizers to reduce the amount of characters used in a textfield.
10. **UsernameAlreadyTaken**: this exception handles the situation when a user/organizer tries to register a new NightPlan account using a username which is already stored in our database.

Each project team member is responsible for handling five exceptions classes:

Nicolas Oberi: DuplicateEventParticipation, GroupAlreadyCreated, InvalidGroupName, InvalidTokenValue, UsernameAlreadyTaken

Matteo Trossi: EventAlreadyAdded, TextTooLongException, EventAlreadyDeleted, InvalidValueException, MinimumAgeException

6 Databases

In the development of our system, we have decided to implement most of the Data Access Objects (DAO) to operate in JDBC (Java Database Connectivity) mode. The decision to use JDBC was driven by several considerations:

I. **Reliability and Performance:** JDBC is a well-established solution for interacting with relational databases, offering high performance and reliability.

II. **Scalability:** Using a relational database like MySQL enables the system to handle an increasing volume of data and requests without compromising performance.

III. **Maintainability:** JDBC provides a standardized method of database access, simplifying maintenance and support.

However, for the Notifications DAO, we have designed it to operate in dual modes: JDBC with MySQL and File System, storing data persistently using the .csv format. This dual implementation was considered to accommodate different use cases and operational flexibility:

→ **JDBC Mode (MySQL):** This mode ensures that notification data is integrated with the rest of the system, taking advantage of MySQL's reliability, performance, and scalability.

→ **File System Mode:** This mode offers a lightweight and easily deployable solution where a full database setup may not be necessary. Notifications are stored in a .csv file, providing a simple and human-readable format that can be used for quick data inspection and migration.

7 SonarCloud

By incorporating SonarCloud into our development workflow, we ensure that our application not only meets functional requirements but also adheres to the highest standards of code quality and security. You can check the results of SonarCloud analysis of this project using the following link:

https://sonarcloud.io/summary/new_code?id=Trossi-Oberi_ispwproject-2324