Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

European Master in Software Engineering

Master Thesis

Design and Deployment of a Social Network Using Microservices on AWS

Author: Jose Luis De Miguel Pérez

June, 2025

Master Thesis

European Master in Software Engineering

Title: Design and Deployment of a Social Network Using Microservices on AWS
June, 2025

Author: Jose Luis De Miguel Pérez

Supervisor:

Angélica de Antonio Jiménez

Associate Professor

Universidad Politécnica de Madrid

Lenguajes y Sistemas Informáticos e Ingeniería de Software

Universidad Politécnica de Madrid

Abstract

This thesis shows the design and deployment of *WhatsThePlan*, a social networking software developed as a practical case study to document modern software engineering practices. The main objective is to serve as a guide for designing, developing, and deploying systems using cloud infrastructure and microservice architecture.

The thesis covers the full software development lifecycle, from requirements specification to cloud deployment, showing how modern architectural patterns and engineering practices can be applied. There is a particular emphasis on the specification and evaluation of quality attributes such as performance, security, maintainability, and usability informed by ISO standards. These are addressed through architectural tactics described in the established industry literature. Every technical decision aligns with clearly defined quality goals. Amazon Web Services (AWS) was selected as the cloud provider, and the thesis details how AWS services were used to deploy the social network in the cloud.

Keywords: *Software Architecture - Cloud Architecture - AWS - Microservices*

Contents

1 Introduction

1.1 Context and Motivation

In recent years, cloud computing has become the standard for building and deploying modern software systems. It allows developers to create scalable, flexible, and reliable applications using managed services, automation, and distributed architectures. At the same time, microservices and DevOps practices have changed how systems are designed and delivered, supporting faster development and easier maintenance.

Social networks are a popular and challenging type of system to build. They require handling many users, real-time updates, and complex interactions between users. For this reason, designing the architecture of a social network is a perfect way to put modern software engineering practices into practice.

This thesis combines my interest in software architecture with the goal of testing my cloud knowledge in a real project. It uses AWS and a microservices approach to create a platform that follows current industry trends and best practices.

1.2 Objectives of the Thesis

The goals of this Master Thesis are as follows:

- Define the problem to be solved, specify functional and non-functional requirements, and translate user needs into user stories to guide the implementation of a social network.
- Design a software architecture based on microservices principles and domain-driven design to ensure the flexibility, fault isolation, and maintainability of the platform.
- Develop the front-end and back-end services of the platform based on the defined architecture and user stories.
- Deploy the system on a cloud infrastructure using Amazon Web Services (AWS), to achieve global distribution.
- Design and implement a Continuous Integration and Continuous Deployment (CI/CD) pipeline using AWS tools to achieve automated delivery of changes.

1.3 Scope and Limitations

This thesis focusses on the architectural design and cloud deployment of a social network using current software engineering practices. The main goal is to build a scalable system using microservices, containerisation, and AWS cloud services while following DevOps and CI/CD principles.

The functional scope of the application is limited to the main features of a minimum viable product (MVP). It includes user registration, event creation and registration, profile management, and reviews. Although the system is technically complete and deployable, it does not include more advanced features such

as a recommendation engine or chat. The focus was on deployment and design and not on a full-featured application.

Since it is an academic work, it has budget constraints, so only AWS free-tier resources were used where possible. This limited the scalability and performance of the system. Although the architecture is designed to support high availability and global reach, the current deployment is not suitable for real-world usage. All services used in this thesis were in the AWS free-tier, which offers basic capabilities but enough for the thesis purpose.

As a result, the system serves as a prototype demonstrating how to apply cloud design patterns in a realistic setting, but it is not production grade in terms of performance or availability.

1.4 Thesis Structure

This thesis is organized into several chapters, each focused on a key aspect of the project.

Chapter 2 contains the project definition. It includes the problem statement, the software requirements specification which has followed standard engineering practices, and user stories with the functionalities of the system. It also describes the development methodology and explains the selection of the technology stack.

Chapter 3 focusses on user interaction and behaviour. Provides a navigation map and defines use cases that describe how different types of user interact with the platform, covering features such as authentication, profile management, event creation, and reviews.

Chapter 4 details the software architecture. It describes the architectural style that has been used, the responsibilities of each component, and the data architecture of the system. It also explains how every quality attribute is addressed through design tactics.

Chapter 5 addresses the cloud infrastructure and the deployment aspects of the system. Describes how AWS services were used to build the cloud infrastructure, covering the presentation, application, data, and communication layers. The chapter also explains the CI/CD pipeline setup and automation strategies.

Chapter 6 explains the results obtained and the author's conclusions. It contains the achieved objectives, the limitations of the work, and future improvements.

2 Project Definition

2.1 Problem Statement

The software system developed in this work, *WhatsThePlan*, is a social networking software that has been designed to connect users through events and activities. It allows users to find and join activities that suit their preferences, with the goal of achieving connections between users.

Although social networks may appear conceptually simple, their implementation presents challenges in modern software engineering. This is because delivering a reliable service to a great number of users requires the system to be scalable, secure, and highly available - which require correct architectural design choices.

This project tries to solve all these complexities by designing and implementing the platform using native principles of the cloud. Cloud computing has changed the way software systems are built, deployed, and scaled, moving away from traditional monolithic and on-premise servers. Some years ago, applications were deployed on a fixed infrastructure with manual provisioning, which had difficult scalability, more complex maintenance, and slower change delivery. In contrast, today, native cloud applications are designed to fully leverage the capabilities of cloud environments: elasticity, on-demand resource provisioning, distributed architectures, and managed services.

Thanks to the cloud native approach used in this project, it benefits from modularity, fault isolation, automatic scaling of individual components, and simplified deployments through containerisation. The cloud deployment of this platform is based on Amazon Web Services (AWS), which offers different services that are easily scaleable. AWS services provide the foundation for building and operating the platform in a secure, resilient, and cost-efficient manner. These tools also collaborate with the realisation of DevOps best practices, such as continuous integration and delivery.

2.2 Software Requirements Specification

This section follows the structure provided by ISO/IEC/IEEE 29148:2018 – Systems and Software Engineering — Life Cycle Processes — Requirements Engineering [1]. This is an international standard that contains guidelines for specifying software requirements, ensuring clarity, consistency, and completeness throughout the system's life cycle. Following these principles guarantees that this specification defines functional and non-functional requirements of the platform in a structured manner, guiding the design and implementation of the system architecture.

2.2.1 Introduction

2.2.1.1 Purpose

This document defines the Software Requirements Specification (SRS) for the *WhatsThePlan* platform. Describes the functional, performance, and interface

requirements to ensure that the application reaches its goals of connecting people through social events.

2.2.1.2 Scope

WhatsThePlan is a social networking platform designed to facilitate connections through events and activities based on user interests and geographic proximity. It includes tools for organizing events and participating in them to enhance the user experience.

2.2.1.3 Product overview

Product perspective

WhatsThePlan operates as a web application. Its user interface is designed for web browsers such as Google Chrome. Communication interfaces include HTTPS protocols for secure data exchange.

Product functions

The main functionalities of the *WhatsThePlan* software system are organized into two primary categories: user and event functions.

- **User Functions:** Users are able to register, create and update their profiles, explore events based on preferences or location, and submit reviews for event organisers they have attended.
- **Event Functions:** Enable users (as organisers) to create, edit, and manage events. Event contains information such as title, date/time, location, and capacity, as well as tracking participant registrations.

User characteristics

The platform is designed for two types of users with the following goals:

- **Individuals seeking social engagement:** Users who want to find and participate in events with their personal interests, with the aim of expanding their social circles and spending quality time.
- **Event organisers:** Users who want tools to create, promote, and track participant registration.

Limitations

The current version of the system has the following limitations:

- **Minimal User Feedback:** Due to limited access to real user testing and validation sessions during development, there might arise usability issues or unanticipated user behaviours.
- **Scalability Trade-offs:** The architecture emphasises modularity, maintainability, and cost efficiency, but certain performance aspects, such as response time under peak load, may not be fully optimised in this version.
- **Absence of a Recommendation System:** Currently all social networks have recommendation systems. For this project, recommendation system is out of scope, so users should do manual exploration of events without personalised guidance.

2.2.1.4 Definitions

This section provides definitions for terms used throughout the document.

- **Event/Activity:** An activity listed on the platform that users can browse, join, or organise. Events could be of different types, such as workshops, meet-ups, sports, or other community activities.
- **Organizer:** A regular user who creates and manages events. Any user with an active account has the ability to act as an organiser by listing events and handling participation.
- **Participant:** A user who joins or registers for an event created by another user or organiser.
- **Platform:** Refers to the entire *WhatsThePlan* system, including its web interface, back-end services, databases, and cloud infrastructure.

2.2.2 References

ISO/IEC/IEEE 29148:2018 – Systems and Software Engineering — Life Cycle Processes — Requirements Engineering [1]

2.2.3 Requirements

2.2.3.1 Functional requirements

User

- **FR-U1**: The system must let users sign up with a working email and password, or by logging in with their Google account.
- **FR-U2**: The system shall validate all registration inputs and authentication attempts, displaying appropriate user-friendly error messages if the provided information is invalid or if Google authentication is unsuccessful.
- **FR-U3**: The system must let users log in with their email and password or through their Google account.
- **FR-U4**: The system shall validate login credentials and authentication attempts, displaying appropriate error messages when the credentials are incorrect or the authentication process is unsuccessful.
- **FR-U5**: The system must allow users to make a personal profile by entering a username, first name, last name, city, and preferences.
- **FR-U6**: The system shall validate profile creation input and ensure that usernames and email addresses are unique, displaying appropriate error messages when validation fails or duplicate information is detected.
- **FR-U7**: The system must let users change their profile information like username, first name, last name, city, and preferences.
- **FR-U8**: The system shall validate profile update inputs, ensuring that all updated fields meet the required format and uniqueness constraints, and display appropriate error messages if validation fails.

Event

- **FR-E1**: The system shall allow users to create events by providing a title, description, date and time, duration, location, capacity, activity types, and image.
- **FR-E2**: The system shall check the event details to make sure all needed information is correct, and show clear error messages if something is wrong or missing.
- **FR-E3**: The system shall notify the user if the event creation process fails and prompt the user to review or retry the submission.
- **FR-E4**: The system shall allow event organizers to update event information, including title, description, date and time, duration, location, capacity, activity types, and image.
- **FR-E5**: The system shall check the updated event information and show error messages if anything is incorrect or missing.
- **FR-E6**: The system shall restrict event updates to the organizer of the event.
- **FR-E7**: The system shall allow users to view detailed information for each event, including the title, description, date and time, duration, location, available capacity, organizer name, activity type, and image.
- **FR-E8**: The system shall tell users when an event is no longer active.
- **FR-E9**: The system shall allow organizers to delete events they have created, removing the event from public listings once deletion is confirmed.
- **FR-E10**: The system shall ask for confirmation before deleting an event, and will cancel the action if the organizer does not confirm.
- **FR-E11**: The system shall restrict event deletion actions to the original organizer of the event.
- **FR-E12**: The system shall let users view a list of events they have signed up for, including event title, date and time, duration, place, and who organized it.
- **FR-E13**: The system shall inform the user when no registered events are available in their history.
- **FR-E14**: The system shall let users view full details of any event from their list of registered events.
- **FR-E15**: The system shall allow users to view a list of events they have organized, including event title, description, date and time, duration, location, and participant count.
- **FR-E16**: The system shall inform users if they have not organized any events.
- **FR-E17**: The system shall let users view full details of any event they have organized, including reviews or feedback.
- **FR-E18**: The system shall allow users to search for events by applying filters such as location, event duration, event date range, and activity type.

- **FR-E19**: The system shall display all upcoming events when no filters are applied by the user.
- **FR-E20**: The system shall tell users if no events match the filters, and suggest changing the search settings.
- **FR-E21**: The system shall allow users to view detailed information for a selected event from the search results.
- **FR-E22**: The system shall allow users to register for an event if there are available spaces, and confirm successful registration.
- **FR-E23**: The system shall stop users from signing up if the event is full, and inform them.
- **FR-E24**: The system shall notify users if they attempt to register for an event they are already registered for.

Review

- **FR-R1**: The system shall allow users who have attended an event to submit a review and a rating to the event organizer.
- **FR-R2**: The system shall check each review to make sure the rating is in the allowed range and the review text follows the rules. If something is wrong, the system shall show a clear error message.
- **FR-R3**: The system shall prevent users from submitting a review for an organizer with whom they have no shared event participation.
- **FR-R4**: The system shall let users read all reviews and ratings given to a specific event organizer.
- **FR-R5**: The system shall inform users when reviews are not available for a specific event organizer.

Notification

- **FR-N1**: The system shall send a welcome email to users right after they create an account. This email shall include a link to the main page of the platform.
- **FR-N2**: The system shall send a registration confirmation email to users upon successfully registering for an event. The email shall include a link to the event details page and to the review submission page.
- **FR-N3**: The system shall send an email to all registered users if an event is cancelled. This email shall include a link to the homepage and suggest checking out other events.

2.2.3.2 Performance Efficiency requirements

- **PR-1:** The system shall show search results in 2 seconds or less for at least 95% of searches during normal use.
- **PR-2:** The system shall return frequently accessed data (such as event details) within 1 second for at least 95% of requests.

- **PR-3:** User-facing operations (registration, event registration, profile updates) shall not be delayed more than 500 milliseconds by background notification tasks.
- **PR-4:** The system shall support up to 500 users doing things at the same time (like searching, registering, or viewing) without getting slower (keeping 95% of actions under 2 seconds).
- **PR-5:** The system shall use pagination for complicated searches, showing no more than 10 results per page to keep response times fast.

2.2.3.3 Interaction Capability (Usability) requirements

- **ICR-1**: The system shall give clear and consistent messages after each user action so users can understand what happened (for example, after submitting or selecting something).
- **ICR-2**: The system shall validate user input as they are entered, where applicable, and present immediate, understandable error messages to guide users in correcting invalid data.
- **ICR-3**: The system shall stop users from making important or final changes by mistake by asking for confirmation before doing those actions.

2.2.3.4 Reliability requirements

Faultlessness

- **RR-1**: The system shall perform all specified functions without faults under normal operation, and shall include mechanisms to verify service availability and correct behaviour automatically.
- **RR-2**: The system shall have tools like logs, metrics, and traces to help find, understand, and fix problems, and reduce the chance of unnoticed errors.

Availability

- **RR-3**: The system shall maintain operational status and be accessible to users at least 99.9% of the time.
- **RR-4**: The system shall give users steady and reliable access during normal usage. It shall avoid slowdowns that could make the service unavailable. The expected usage levels will be reviewed regularly as part of system upkeep.
- **RR-5**: The system shall maintain service availability by detecting and isolating faulty components, ensuring users are not impacted by individual component failures.

Fault Tolerance

- **RR-6**: The system shall keep running and handle requests even if one of its parts fails.
- **RR-7**: The system shall detect when a component stops working and avoid sending requests to it, using healthy components instead.

Recoverability

- **RR-8**: The system shall restart any part that fails automatically, without needing someone to do it manually.
- **RR-9**: The system shall recover normal operation after an interruption within a maximum of 5 minutes.

2.2.3.5 Security requirements

- **SR-1**: The system shall prevent unauthorized users from accessing sensitive data and system features.
- **SR-2**: The system shall make sure that only users with the correct roles and permissions can do restricted tasks.
- **SR-3**: The system shall verify user identity before granting access to sensitive data or restricted actions.
- **SR-4**: All communication between the user and the server shall be protected with HTTPS to keep user data safe.
- **SR-5**: The system shall use session time limits and token updates to stop others from getting into user accounts without permission.
- **SR-6**: The system shall store passwords safely by using secure methods like hashing and salting to protect them from attacks.

2.2.3.6 Maintainability requirements

Modularity

- **MR-1**: The system shall be built so that changes in one part do not affect other parts that are not related.
- **MR-2**: Different parts of the system must have clearly separated responsibilities, with minimal coupling between them.

Modifiability

- **MR-3**: It shall be easy to change the database structure without stopping the system or causing errors.
- **MR-4**: Each item in the database shall include data showing when it was created and last updated, to help track changes and fix problems.
- **MR-5**: The system shall allow configuration values to be easily changed or updated without requiring recompilation or redeployment, while ensuring these values are stored and managed securely.

Testability

- **MR-6**: Each part of the system must be independently testable without needing the whole system to be running.
- **MR-7**: The system shall support quick checks, using automated tests, to find out if a change has caused any problems.

2.2.3.7 Flexibility requirements

Adaptability

- **FLR-1**: The product shall be packaged for consistent deployment.

Scalability

- **FLR-2**: The system shall automatically use more or fewer resources depending on the current workload, so that performance and availability stay stable without needing someone to make manual changes.
- **FLR-3**: The product shall avoid single points of failure to ensure global availability and stable performance.

Installability

- **FLR-4**: The deployment process shall be automated to reduce manual steps and human error.
- **FLR-5**: The system shall allow quick rollback to a previous stable version if a deployment fails or has errors.
- **FLR-6**: The deployment process shall minimize downtime during updates and rollbacks.

2.3 User Stories

User stories help us understand what users want to do with a system by describing simple, clear situations from their point of view. They explain the main tasks and features the platform should have—like signing up, creating events, writing reviews, and receiving notifications. This user-centered approach guides development to ensure the system works well for real people. User stories first became popular in the early 2000s through agile development methods like Scrum and Extreme Programming, thanks to Mike Cohn [2]. Unlike traditional, heavy requirements documents, user stories are short and informal, focusing on what users need. They support teamwork, flexibility, and delivering value step by step. They follow a simple format: "As a [user role], I want [goal] so that [benefit]," helping teams keep attention on real user needs and connect those needs with the technical design of the platform.

2.3.1 Users

User Story: User Registration (US-1)

As a user **I want** to create an account using either my email and password or my Google account **so that** I can access the platform and personalize my experience.

Acceptance Criteria:

- **Given** the registration page,
  when the user enters valid details (username, email, password),
  then then my account should be created successfully and handled by AWS Cognito.
- **Given** the registration page,
  when the user chooses to register using their Google account and gives permission,
  then my account should also be created successfully and handled by AWS Cognito.
- **Given** invalid input during email/password registration (e.g., invalid

email format, weak password),
when the user submits the form,
then an appropriate error message should be displayed guiding the user to correct the input.
- **Given** a problem during Google login (e.g., access denied or login error),
when the user tries to register with Google,
then the system should inform the user the registration failed and offer to try again or use another way.

Estimation: 3 Story Points
Functional Requirements: FR-U1, FR-U2
Additional Notes: User registration is managed securely using AWS Cognito. It supports both regular email/password and Google sign-up. All identity checks and data handling are done securely by Cognito.

User Story: User Login (US-2)

As a user **I want** to log in to the platform using my email and password or my Google account **so that** I can safely access my personalized experience.

Acceptance Criteria:
- **Given** the login page,
when the user types in the correct email and password,
then the system should log them in and give access to the platform.
- **Given** the login page,
when the user chooses to log in with their Google account and allows access,
then the system should log them in and give access to the platform.
- **Given** incorrect login details (like a wrong email or password),
when the user tries to log in,
then the system should show a clear message asking the user to check and fix the login information.
- **Given** a problem with Google login (such as access denied or failed sign-in),
when the user tries to log in using Google,
then the system should show a message saying login didn't work and suggest trying again or using another method.

Estimation: 3 Story Points
Functional Requirements: FR-U3, FR-U4
Additional Notes: The login process is safely managed using AWS Cognito. Users can log in with their email and password or sign in through Google. Cognito handles all sessions and identity checks securely.

User Story: Profile Creation (US-3)

As a user **I want** to create my profile **so that** I can personalize my experience.

Acceptance Criteria:
- **Given** the profile creation form,
when the user provides valid details (username, first name, last name, city, and preferences),

then the profile should be successfully created and saved.
- **Given** a username that is already taken,
  when the user submits the form,
  then the system should inform the user that the username is unavailable and request a new one.
- **Given** missing or invalid input (such as blank fields, or an incorrectly formatted username),
  when the user submits the form,
  then the system should display validation error messages explaining what needs to be corrected.

Estimation: 3 Story Points
Functional Requirements: FR-U5, FR-U6
Additional Notes: The profile creation form enforces strict validation rules for all fields, including username uniqueness and proper formatting. Preferences are optional.

User Story: Profile Update (US-4)

As a user **I want** to edit my profile **so that** my information stays up to date and matches my interests.

Acceptance Criteria:
- **Given** the profile update form,
  when the user enters valid new details (like a different username, updated name, city, or preferences),
  then the system should update the profile and show the new information.
- **Given** the user tries to change the username to one already in use,
  when they submit the update,
  then the system should tell the user the name is taken and ask for another one.
- **Given** wrong or missing input (like empty fields or invalid formats),
  when the user submits the form,
  then the system should show clear messages explaining what needs to be fixed.

Estimation: 3 Story Points
Functional Requirements: FR-U7, FR-U8
Additional Notes: Profile updates follow the same rules as when the profile was first created. The system checks for correct and unique data. Only users who are logged in can change their profile.

2.3.2 Events

User Story: Event Creation (US-5)

As a user **I want** to create events **so that** others can discover them and join activities I organize.

Acceptance Criteria:
- **Given** the event creation form,
  when the user provides valid details (such as title, description, date and

time, duration, location, capacity, and activity types),
then the event should be successfully created and made available for others to view and join.
- **Given** missing or invalid information (such as an empty title, past event date, missing duration, or a capacity lower than one),
when the user submits the event creation form,
then the system should display clear validation error messages guiding the user to correct the input.

Estimation: 5 Story Points
Functional Requirements: FR-E1, FR-E2, FR-E3
Additional Notes: The event creation process ensures that all mandatory fields are properly validated. Events must have meaningful information and a valid future date to be published.

User Story: Event Update (US-6)

As a user **I want** to edit events I have created **so that** I can fix mistakes or change the event details if needed.

Acceptance Criteria:
- **Given** that the user is the event organizer,
when they update event details (like title, description, date and time, duration, location, capacity, activity type, or image),
then the system should save the new information and show it in the event listing.
- **Given** invalid or incomplete input (such as missing title, past date, invalid duration, or a capacity lower than allowed),
when the user submits the update form,
then the system should display clear error messages prompting the user to correct the information.
- **Given** that the user is the event organizer and the event has already passed,
when they try to update the event,
then the system should reject the update and display a message that editing past events is not allowed.

Estimation: 5 Story Points
Functional Requirements: FR-E4, FR-E5, FR-E6
Additional Notes: Only the person who created the event can make updates. This helps make sure that event info is correct and stays up to date for all users.

User Story: View Event Details (US-7)

As a user **I want** to view detailed event information **so that** I can decide whether I want to participate.

Acceptance Criteria:
- **Given** the list of available events,
when the user selects an event,
then the system should display detailed information including the title, description, date and time, location, available capacity, organizer name,

and activity type.
- **Given** that the user is viewing the event details,
  when the event has additional information (such as images or categories),
  then the system should also present this information clearly to help the user make an informed decision.
- **Given** that an event is no longer available (deleted or completed),
  when the user tries to access its details,
  then the system should inform the user that the event is no longer active.

Estimation: 3 Story Points
Functional Requirements: FR-E7, FR-E8
Additional Notes: The event details page provides all necessary information for users to evaluate whether they want to attend an event. Additional event media or activity types may enhance user decision-making but are optional.

User Story: Deleting an Event (US-8)

As an organizer **I want** to delete events I created **so that** I can remove events I no longer need.

Acceptance Criteria:
- **Given** the list of events made by the organizer,
  when the organizer chooses one to delete and confirms it,
  then the event should be removed from the platform and not shown to users any more.
- **Given** that a user is looking at events,
  when an event has been deleted by its organizer,
  then it should no longer appear in search results or event lists.
- **Given** that the organizer starts to delete an event,
  when they cancel the confirmation step,
  then the event should not be deleted and should still be active.
- **Given** that the organizer tries to delete an event that has already passed,
  when they confirm the deletion,
  then the system should reject the deletion and display a message that deleting past events is not allowed.

Estimation: 3 Story Points
Functional Requirements: FR-E9, FR-E10, FR-E11
Additional Notes: Only the organizer who created the event can delete it. Once deleted, the event will be hidden from all users and removed from the platform.

User Story: View My Registered Events (US-9)

As a user **I want** to see the events I signed up for **so that** I can check past or upcoming events and keep track of what I've done.

Acceptance Criteria:
- **Given** that the user has registered for events,
  when they go to the 'My Registrations' section,
  then a list of all their registered events should appear, showing details

like the event name, date, place, and organizer.
- **Given** that the user has not yet joined any events,
  when they open the 'My Registrations' section,
  then the system should show a message saying there are no registered events.
- **Given** the list of events,
  when the user clicks on one event,
  then the system should show full details about that event.

Estimation: 3 Story Points
Functional Requirements: FR-E12, FR-E13, FR-E14
Additional Notes: The 'My Registrations' section helps users see which events they have joined, making it easy to revisit or follow up on past and future activities.

User Story: View Organized Events (US-10)

As a user **I want** to see the events I have created **so that** I can check their details and keep track of what I've organized.

Acceptance Criteria:
- **Given** that the user has created events,
  when they open the 'My Events' section,
  then the system should show a list of all their events, including the title, description, date, location, and number of participants.
- **Given** that the user has not created any events,
  when they go to the 'My Events' section,
  then the system should show a message saying there are no events to display.
- **Given** the list of events,
  when the user selects one event,
  then the system should show full details about it, including any feedback or reviews it received.

Estimation: 3 Story Points
Functional Requirements: FR-E15, FR-E16, FR-E17
Additional Notes: The 'My Events' section helps users view and manage the events they have organized, giving them a simple way to track their activity.

User Story: Event Search (US-11)

As a user **I want** to search for events **so that** I can find activities that match my interests, location, and availability.

Acceptance Criteria:
- **Given** the event search page,
  when the user uses filters like location, event length, date range, or type of activity,
  then the system should show a list of events that fit the selected options.
- **Given** that the user does not use any filters,
  when they start a search,
  then the system should show all upcoming events by default.

- **Given** that no events match the filters,
  when the user searches,
  then the system should show a message saying no events were found and suggest changing the filters.
- **Given** a list of search results,
  when the user clicks on an event,
  then the system should show full details about that event.

Estimation: 3 Story Points
Functional Requirements: FR-E18, FR-E19, FR-E20, FR-E21
Additional Notes: The search feature helps users quickly find events that match their needs. Filters like place, time, duration, and activity type make the results more useful and easier to explore.

User Story: Event Registration (US-12)

As a user **I want** to register for events **so that** I can participate in activities that interest me.

Acceptance Criteria:
- **Given** an event with available spaces,
  when the user clicks the 'Register' button and confirms their participation,
  then the system should successfully register the user for the event and confirm their spot.
- **Given** that the event has already reached full capacity,
  when the user attempts to register,
  then the system should inform the user that the event is full and they cannot register.
- **Given** that the user is already registered for an event,
  when the user tries to register again,
  then the system should notify the user that they are already enrolled in the event.

Estimation: 3 Story Points
Functional Requirements: FR-E22, FR-E23, FR-E24
Additional Notes: The event registration process ensures users can secure their place in an activity, while preventing overbooking or duplicate registrations. Users receive immediate feedback about their registration status.

2.3.3 Reviews

User Story: Submit a Review (US-13)

As a user **I want** to write a review and give a rating to the organizer of an event I joined **so that** I can share my opinion and help others know if the event was good.

Acceptance Criteria:
- **Given** that the user joined an event organized by someone else,
  when they send a review with a rating and a message,
  then the system should save the review and link it to the organizer.

- **Given** that the review text is empty or too long,
  when the user tries to send it,
  then the system should show a clear message asking the user to fix the input.

Estimation: 5 Story Points
Functional Requirements: FR-R1, FR-R2
Additional Notes: Only users who joined an event can write a review. Ratings must be from 1 to 5, and the message must be short. This helps keep reviews useful and fair for everyone.

User Story: View Organizer Reviews (US-14)

As a user **I want** to read reviews about an event organizer **so that** I can decide if I can trust them before joining their events.

Acceptance Criteria:
- **Given** an event with reviews,
  when the user opens the reviews section in event details page,
  then the system should show all reviews, including ratings and comments from people who joined past events.
- **Given** that no reviews have been written yet,
  when the user checks the reviews section,
  then the system should tell the user that there is no feedback available.
- **Given** a list of reviews,
  when the user reads through them,
  then the system should clearly show each review along with the organizer's average rating.

Estimation: 3 Story Points
Functional Requirements: FR-R3, FR-R4, FR-R5
Additional Notes: The reviews section helps users choose events wisely by showing honest feedback from others. If there are no reviews, the system will show a message explaining that.

2.3.4 Notifications

User Story: Send Welcome Email (US-15)

As a new user **I want** to receive a welcome email after creating my account **so that** I feel acknowledged and can easily access the platform.

Acceptance Criteria:
- **Given** that a user has successfully created an account,
  when the registration process is completed,
  then the system shall send a welcome email to the user including a link to the homepage.

Estimation: 2 Story Points
Functional Requirements: FR-N1
Additional Notes: The welcome email enhances user onboarding and should be sent promptly upon account creation.

User Story: Send Event Registration Confirmation Email (US-16)

As a user **I want** to get a confirmation email after signing up for an event **so that** I can check the event details and leave a review later.

Acceptance Criteria:
- **Given** that the user has signed up for an event,
  when the registration is complete,
  then the system should send a confirmation email with a link to the event page and the review page.

Estimation: 3 Story Points
Functional Requirements: FR-N2
Additional Notes: The confirmation email should include useful links and event information to help users stay involved and give feedback.

User Story: Send Event Cancellation Email (US-17)

As a user **I want** to get an email if an event I signed up for is cancelled **so that** I can change my plans and look for other events.

Acceptance Criteria:
- **Given** that an event is cancelled,
  when the cancellation is confirmed,
  then the system should send an email to everyone who registered, telling them the event was cancelled and sharing a link to the homepage to explore more events.

Estimation: 3 Story Points
Functional Requirements: FR-N3
Additional Notes: The cancellation email should be easy to understand and invite users to return to the platform and stay engaged.

2.4 Project development

Once having correctly defined the requirements and scope in section 2.2 [Software Requirements Specification], the next step is to begin the design and implementation of the software system. In this section, the process of building the software system given the SRS will be outlined.

2.4.1 Software Architecture

The base of the software system is its architecture, which shows how the system is organized, what the main parts are, and how they work together. This software architecture comes from studying both functional and non-functional requirements, and it turns these needs into a clear set of design choices that help guide how the system is built and used.

Because the software architecture sets up the system's structure and affects which technologies are used, how the system is deployed, and how development is done, it is an important guide during the whole project.

This step will be covered in Section 4 [Software Architecture].

2.4.2 Technology Stack Selection

With the software architecture in place, the next step is to determine the technology stack for development. That is, to select the potential options for technologies used throughout application development. When determining which technologies will be used, the following must be considered: compatibility with architectural style, ability to meet functional and non-functional requirements, maturity of technology, stability, and the associated ecosystem and community support as well as team experience and comfort level.

The goal is to ensure that technology will not only adhere to the desired architecture but also be developed comfortably, efficiently, and ensure successful deployment and production usage. The following subsections explore the chosen technologies for each portion of the system and rationales for selection.

2.4.2.1 Application Frameworks

The back-end and front-end frameworks are key parts of the software system because they define how the logic of the app works and how users use the platform. These frameworks were chosen to match the microservices-based architecture created during the design phase and to make sure the system is scalable, easy to maintain, and simple to develop.

Back-end framework: Spring Boot

Spring Boot [3], which is built on the Java programming language, was chosen for the back end for several reasons. It works well with microservice architectures, which was an important part of this system's design. Spring Boot makes it easier to build small and separate services. It also supports RESTful APIs, helping to set up clear ways for services to talk to each other and to the front end.

Spring Boot also includes ready-to-use tools for things like checking input, handling security, managing dependencies, and working with data. Since it is open source and has strong community support, it is a trustworthy and long-term choice. Java is also a popular language with many tools and learning resources, which helps make the system easier to maintain and gives access to skilled developers.

Front-end framework: Angular

Angular [4], originally developed by Google and actively maintained as an open source project, was chosen as the front-end framework because of its suitability to develop single-page applications (SPA). Angular provides a comprehensive set of tools and features that support the development of a dynamic and responsive user interface.

Furthermore, Angular is built on TypeScript, which introduces strong type and object-oriented features, enhancing code maintainability, and reducing the likelihood of runtime errors. Its worldwide adoption and ecosystem offer access to various libraries and integrations, facilitating faster development and easier maintenance over time.

2.4.2.2 Databases and Storage

The platform requires multiple types of data storage to store the different information it handles. Different access patterns and data formats require a specific approach for every case. As a result, a set of specialised storage system was adopted, each serving a distinct purpose within the system.

PostgreSQL

For structured and relational data, PostgreSQL [5] was the choice. This open source relational database management system is widely used, supports the SQL language, and offers strong transactional guarantees. These characteristics make it particularly suitable for data that demand consistency and integrity.

MongoDB

To handle more flexible and semi-structured information, MongoDB [6] was introduced. As a NoSQL document-oriented database, MongoDB allows for evolving data structures, making it a practical option when schemas may change.

Redis

Redis [7] is an in-memory data store, that is primarily used to cache frequently accessed information. Its ability to deliver fast read and write operations significantly improves response times and reduces the load on primary databases during high demand.

Amazon S3

Amazon Simple Storage Service (S3) [8] is used to store static elements such as images. As an object storage service on the cloud, S3 offers scalability and durability, ensuring that large binary files can be stored and served efficiently.

2.4.2.3 Communication

Reliable communication between all parts of a distributed system is very important. In this platform, a microservices architecture is used, so different services need to exchange data clearly and quickly. Depending on the situation, they may interact directly (synchronously) or send messages to be read later (asynchronously). To support these communication styles, two methods were chosen: HTTP is used for direct request-response communication, and RabbitMQ is used for asynchronous messaging.

HTTP

HTTP serves as the foundation for synchronous communication between services and between the front-end and back-end. Provides a widely adopted language-agnostic protocol for building RESTful APIs, which allows services to expose well-defined interfaces for data access and command execution. Its simplicity and ubiquity make it an ideal default choice for service-to-service interaction when immediate responses are expected.

RabbitMQ

But not all types of communication work well with the request-response method. In some cases, it is better to keep parts of the system more separate, to react to events, or to allow updates to happen later. For these situations, RabbitMQ [9] is very useful. It is a trusted open-source message broker that lets services send

and receive messages asynchronously. This supports background processing, reduces how much system parts depend on each other, and helps the system stay stable under different loads.

2.4.2.4 Cloud Provider

To run and manage the platform, Amazon Web Services (AWS) [10] was chosen as the cloud provider. AWS provides many services that match the system's needs, such as the ability to scale, stay reliable, and serve users globally.

In early 2025, AWS held a leading position in the cloud infrastructure market with a 31% market share. This placed it ahead of major providers like Microsoft Azure and Google Cloud Platform [11]. With its wide network of data centres and availability zones, AWS can offer fast and reliable service to users around the world.

AWS was also selected because of its strong set of tools and constant improvements. These tools help developers build, deploy, and manage apps more easily. This makes AWS a strong choice for hosting the platform and its resources.

2.4.3 Cloud Architecture

The deployment and operational aspects of the system are defined by its cloud architecture, which specifies how the application components are hosted, scaled, and managed within the cloud environment. This architecture addresses infrastructure level concerns such as availability and scalability, ensuring that the platform can operate reliably under real-world usage conditions.

A detailed explanation of the cloud infrastructure and its deployment model, including the use of AWS services and architectural tactics at the environment level, is provided in Section 5 [Cloud Architecture].

2.4.4 CI/CD Pipeline Setup

Setting up a Continuous Integration and Continuous Deployment (CI/CD) pipeline is very important for making software updates fast and automatic. It helps move code from development to deployment more smoothly and reduces the manual effort needed to release new features or changes.

In this project, the CI/CD pipeline is a central part of the delivery process. It supports a method where each approved code change is deployed directly to the production environment. This allows for quick updates, faster release cycles, and better development productivity.

A full explanation of the CI/CD design, the tools used, and how the pipeline connects to the cloud infrastructure is given in Section 5.2 [Continuous Integration and Deployment Pipeline].

2.4.5 Application Development

Once having defined the architecture, technology stack, cloud infrastructure, and delivery pipeline, it take place the development phase, which focusses on implementing the system's functional requirements just as they are described in the user stories. Each story represents a specific feature or behaviour expected from the platform and serves as the primary unit of work throughout this phase.

Back-end and front-end are developed together, with close coordination to ensure consistency in how functionality is implemented and exposed across layers.

In general, application development marks the practical realisation of the system design, transforming abstract specifications into a working software product ready for validation and deployment.

2.4.6 Verification and Validation

Verification and validation ensure that the system behaves as expected. Also, that each implemented feature satisfies its corresponding requirements. This phase is very important to detect errors early, prevent regressions, and ensure the reliability of the platform before and after deployment.

Testing is done automatically in the CI/CD pipeline. Unit tests are written to verify individual components on its own, focussing on the correctness of business logic, data handling, and edge conditions. These tests run automatically during the build process to provide immediate feedback on code changes.

Integration tests are used to validate the interactions between components, so that it can be ensure that the services function correctly together. These tests are executed also automatically in the CI/CD pipeline. This helps to detect inconsistencies or mismatches between services.

Together, these testing methods contribute to the quality of the system, which adds confidence and collaborates with the continuous delivery into the cloud.

3 User Interaction

This chapter connects the system requirements (explained in Section 2.2 [Software Requirements Specification]) with the technical design shown in subsequent chapters. Describes how users interact with the platform by showing the navigation flow and describing the main actions through clear use cases.

The Navigation Map provides a visual overview of the structure of the platform's interface and user transitions, forming a high-level mental model of the interactive logic of the application. The Use Cases then detail the dynamic interactions users perform within this structure, translating user stories and functional requirements into step-by-step scenarios. Together, these elements establish a foundation that directly informs architectural decisions and system design in the following chapters.

3.1 Navigation Map

Knowing how users move through the system and use its features is important for checking if the design fits real needs. Before showing the use cases, this section presents a general navigation map that shows the main web pages, user movements, and key messages or actions triggered along the way.

The diagram 3.1 [Navigation flow of the WhatsThePlan web application] highlights important screens, such as sign-up, log-in, event creation, and feedback. It also shows how users move between these pages, either through the website or through email links. This map sets the stage for the more detailed use cases that come next.

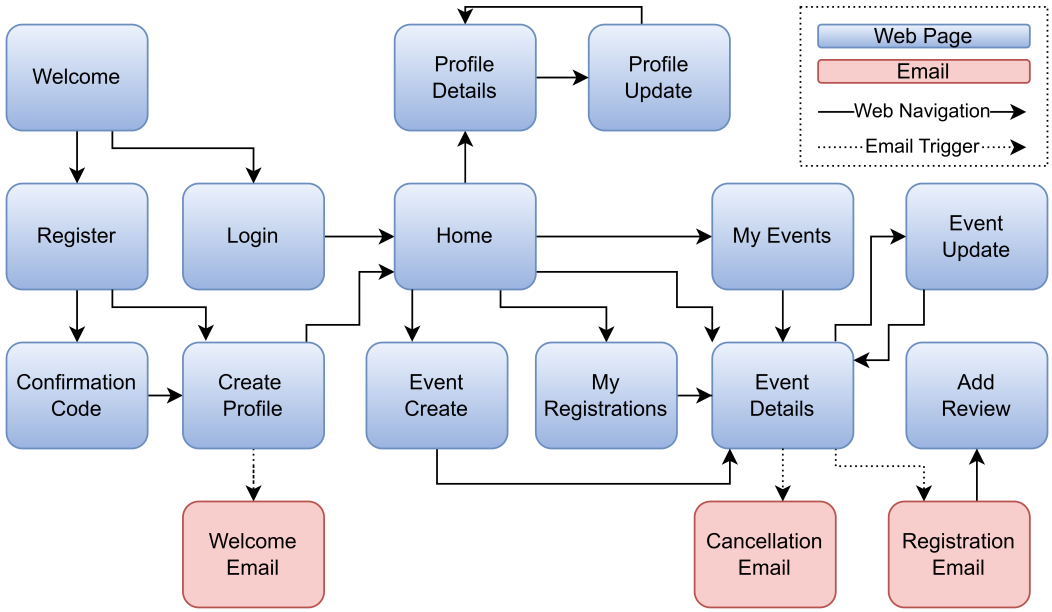


Figure 3.1: Navigation flow of the WhatsThePlan web application

3.2 Use Cases

Use cases are essential for translating high-level user stories into precise, actionable system behaviour. While user stories describe what users want to achieve, use cases define how the system should respond through structured, scenario-driven interactions. This formalisation ensures that functional requirements become concrete, testable, and aligned with user expectations.

Each use case in this section comes from one or more user stories and shows an important way in which users interact with the platform. These use cases help explain what the system should do, guide design choices, and define the roles, limits, and possible errors in each scenario. This directly affects the way services, data models, and APIs are built.

The use cases are based on the navigation structure shown in Figure 3.1 [Navigation flow of the WhatsThePlan web application], which was introduced earlier. That figure shows the main pages, user paths, and key messages in the system. It provides context for the step-by-step scenarios that follow.

3.2.1 User registration and Profile creation

Use Case 1: User registration and Profile creation

Actor(s): User

Goal: User registration and Profile creation

Preconditions: User is not yet registered and is on the welcome page.

Postconditions: User has a verified account, a completed profile, and is redirected to the home dashboard.

Initial state: User is on the welcome screen and chooses to join the platform.

Main Flow (Happy Path):

1. User selects “Get started” button.
2. System displays registration form.
3. User fills in email, password, and password confirmation.
4. User submits the form.
5. System validates input and checks uniqueness.
6. System sends confirmation code to user’s email.
7. User enters confirmation code.
8. System verifies code and creates account.
9. User is redirected to profile creation form.
10. User enters username, name, city, and preferences.
11. System validates and stores profile.
12. System sends welcome email.
13. User is redirected to home.

Alternative Flows:

3b. *User chooses Google registration instead of email/password:*
- 3b1. System redirects to Google OAuth consent screen.
- 3b2. User authorizes access and Google returns identity info.
- 3b3. System skips steps 4–7 and resumes at step 8.

Exception Flows (Unhappy Paths):

- At step 5: Registration input validation fails :
  - Email already exists → system shows duplication error and returns to step 3.
  - Password too weak → system displays password requirements and returns to step 3.
- At step 7: Confirmation code error:
  - Code is invalid or expired → system prompts re-entry or resend and loops back to step 7.
- At step 11: Profile creation input validation fails:
  - Missing or malformed fields → system highlights issues and returns to step 10.
  - Username already taken → system prompts for a different value and returns to step 10.
- At step 3b2: Google registration fails:
  - OAuth error or consent denied → system displays failure message and returns user to step 1.

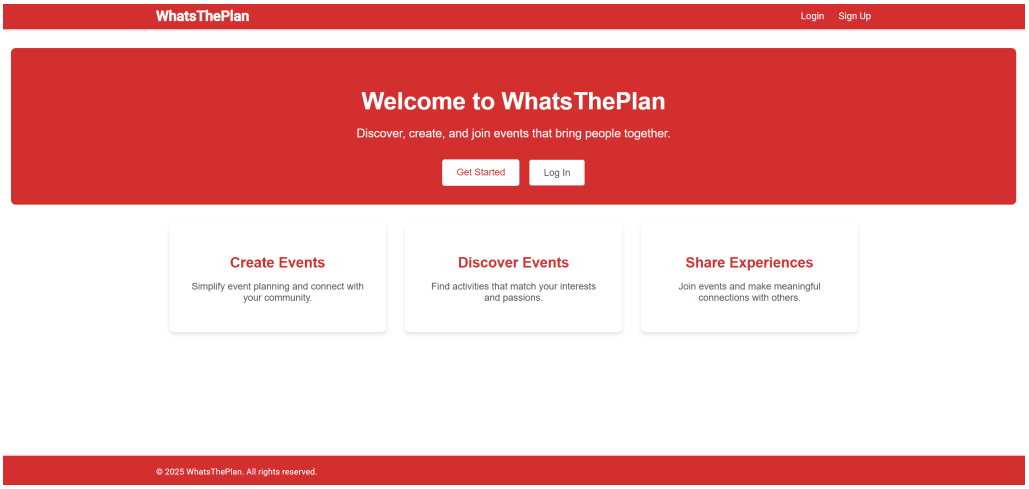


Figure 3.2: Welcome page

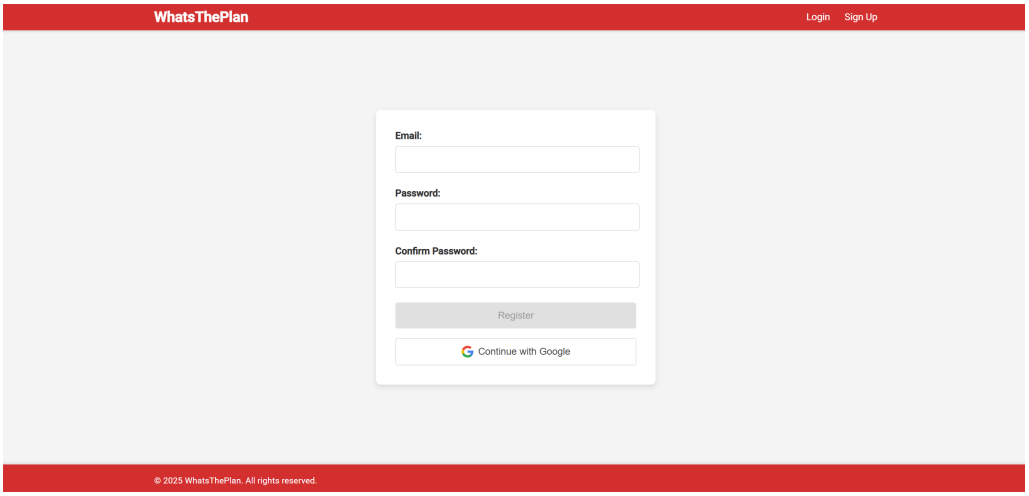


Figure 3.3: Register page

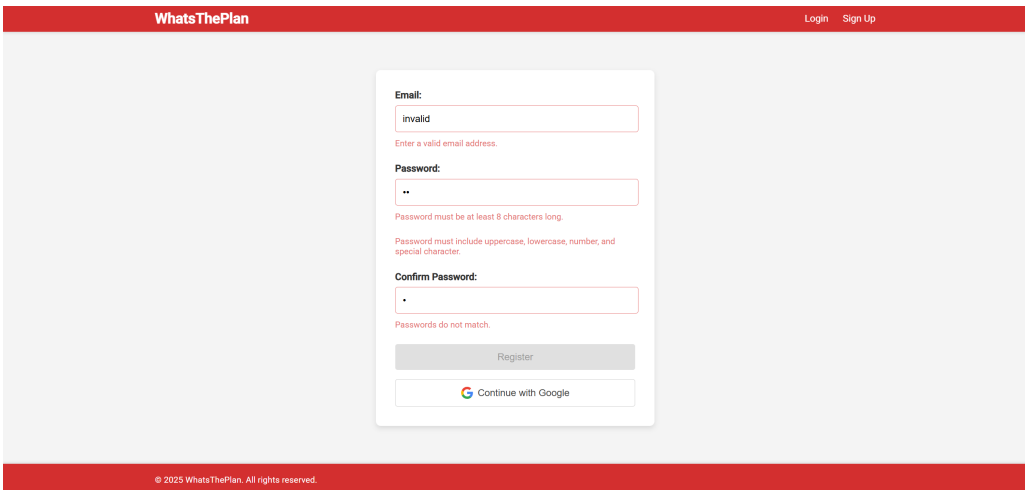


Figure 3.4: Register page with invalid input error messages

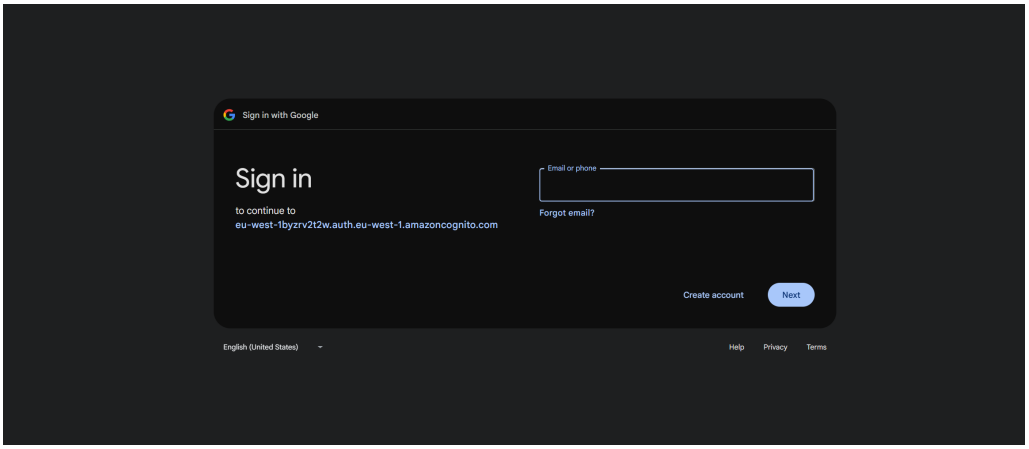


Figure 3.5: Google authentication page

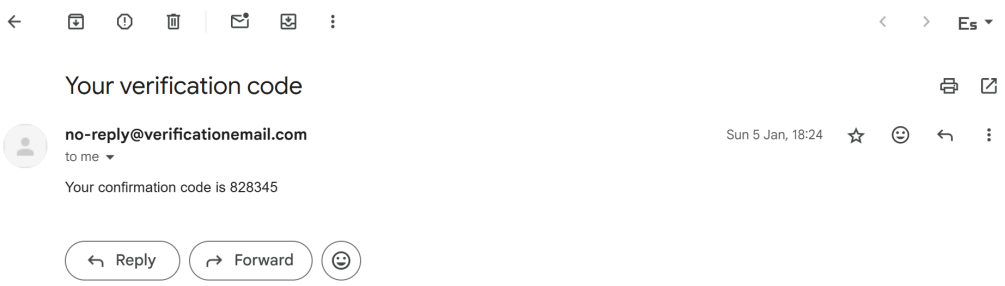


Figure 3.6: Confirmation code email

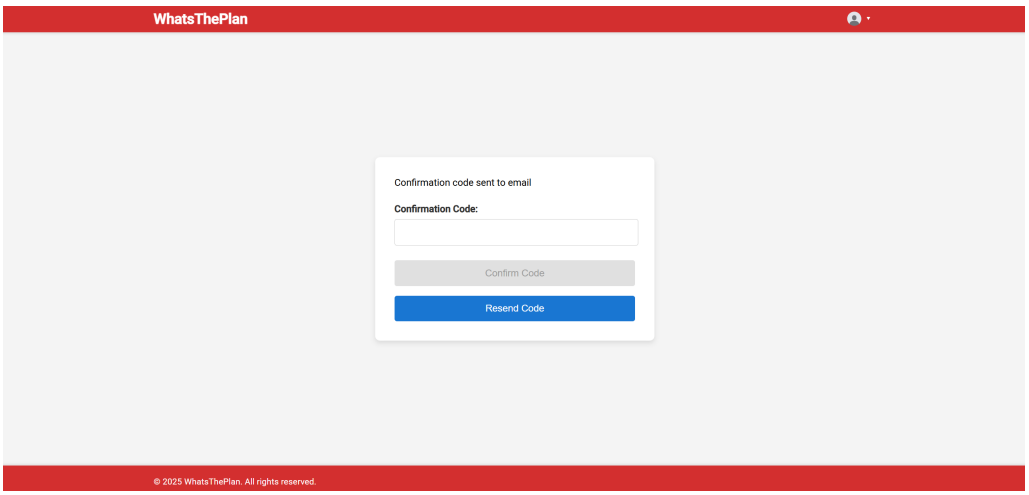


Figure 3.7: Confirmation code page

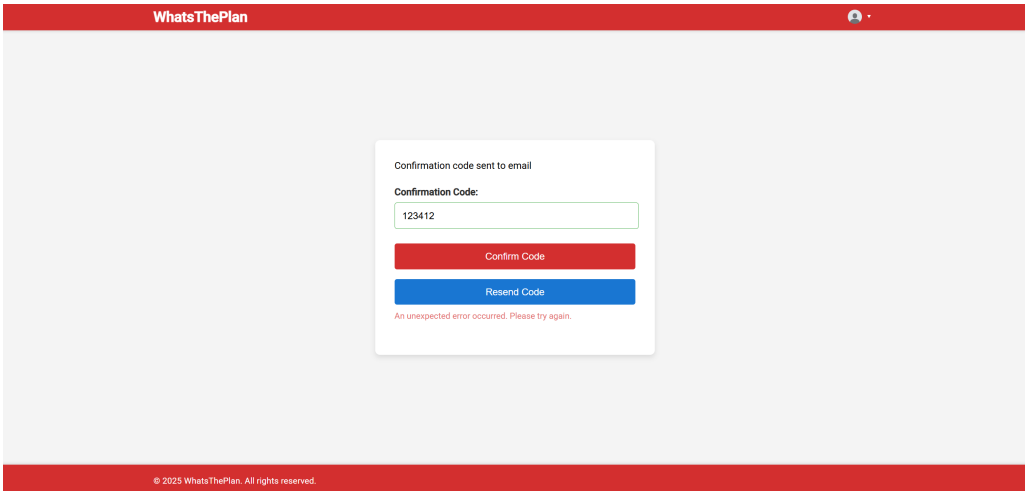


Figure 3.8: Confirmation code page with error message

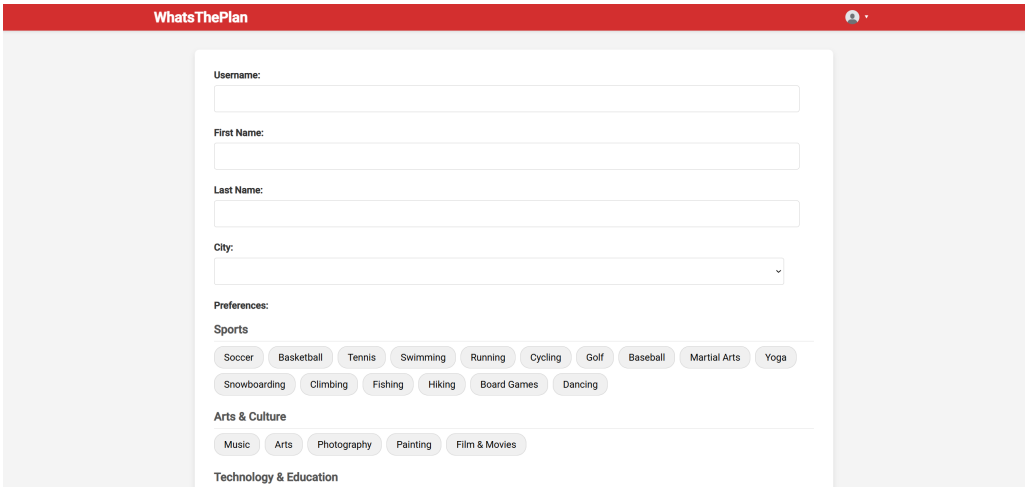


Figure 3.9: Complete profile page

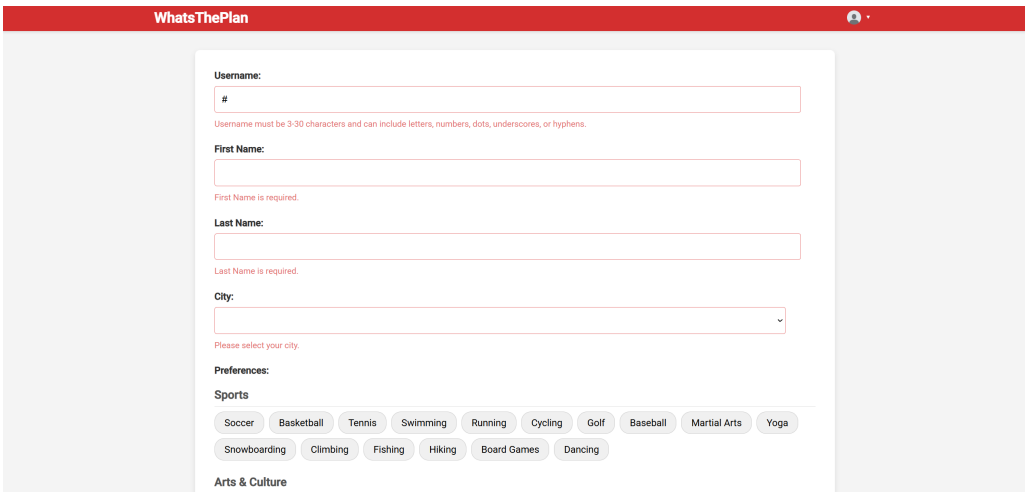


Figure 3.10: Complete profile page with invalid input error messages

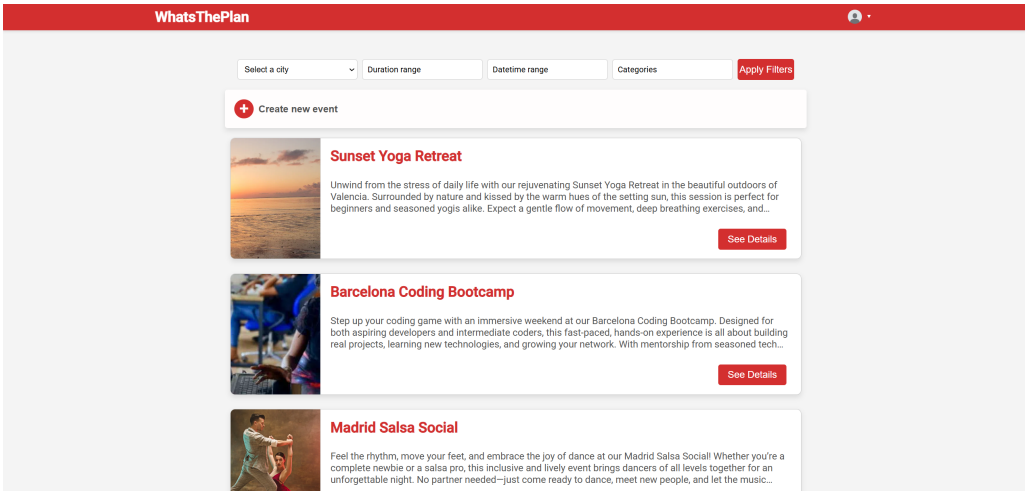


Figure 3.11: Home page

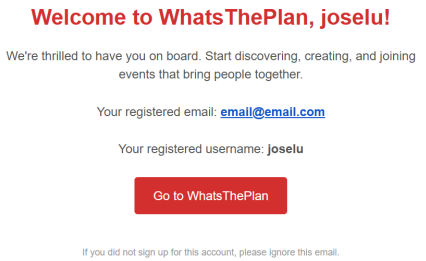


Figure 3.12: Welcome email users receive on registration

3.2.2 User Login and Access

Use Case 2: User Login and Access

Actor(s): User

Goal: User Login and Access

Preconditions: User has an existing account and is not logged in.

Postconditions: User is authenticated and lands on the Home page with access to personalized content.

Initial state: User selects the "Login" option from the Welcome page or navigates directly to Login page.

Main Flow (Happy Path):

1. User accesses the Login page.
2. User chooses authentication method: email/password.
3. System displays the corresponding login form.
4. User enters credentials or completes Google login.
5. System validates credentials and confirms authentication.
6. System redirects user to the Home page, offering accesses to personalized features: Profile Details, Event Create, My Registrations, My Events.

Alternative Flows:

2b. Alternative login via Google:
- 2b1. User selects "Sign in with Google".
- 2b2. System redirects to Google OAuth consent screen.
- 2b3. User authorizes access and returns to step 5.

Exception Flows (Unhappy Paths):

- At step 5: Incorrect email or password:
  - System shows "Incorrect username or password." message and returns to step 2.
- At step 2b3: Google login fails:
  - OAuth rejection or network error → system notifies user and suggests retry or alternative method.

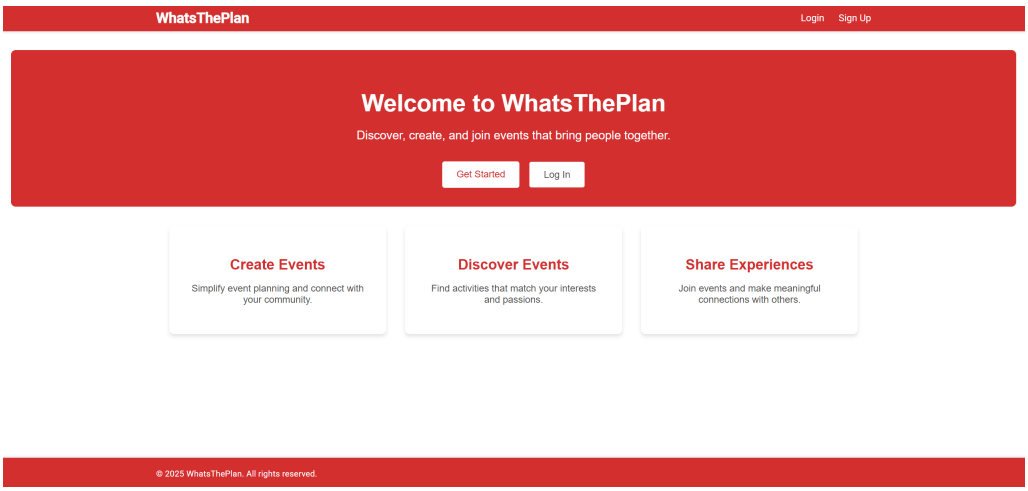


Figure 3.13: Welcome page

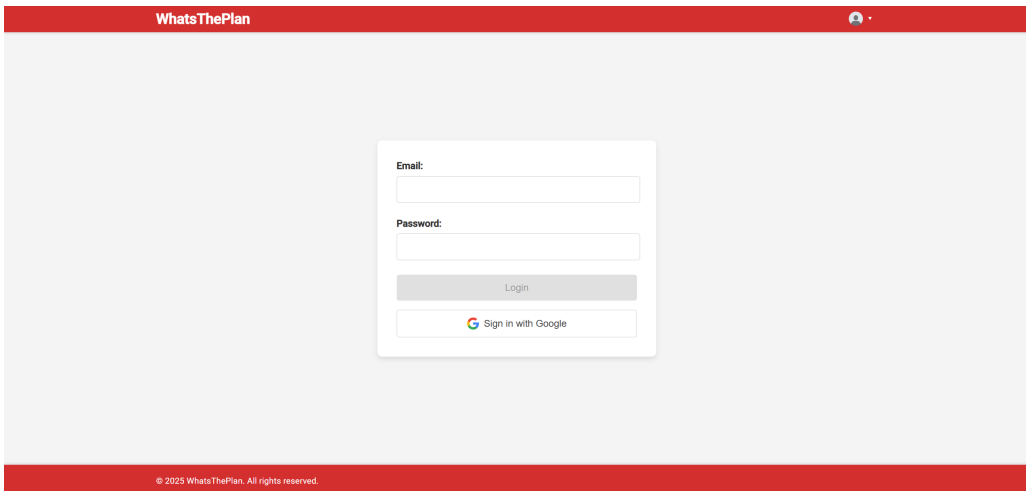


Figure 3.14: Login page

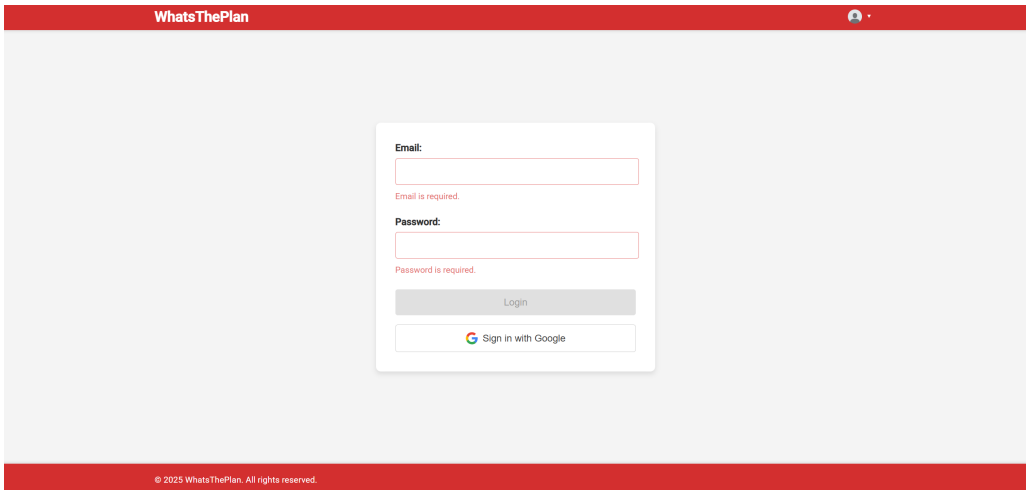


Figure 3.15: Login page with invalid input error messages

3.2.3 User Profile Management

Use Case 3: User Profile Management

Actor(s): User

Goal: User Profile Management

Preconditions: User is authenticated and is on the Home page.

Postconditions: User's updated profile is saved, and the user is redirected back to Profile Details.

Initial state: User selects "Profile" from the Home page.

Main Flow (Happy Path):

1. User selects "Profile" from the Home page.
2. System displays the current profile information.
3. User clicks "Edit Profile".
4. System displays the Profile Update form with pre-filled values.
5. User updates one or more fields (username, name, city, preferences).

6. User submits the form.
7. System validates the input.
8. System updates the profile in the database.
9. System redirects user back to Profile Details view with updated data.

Alternative Flows:

Exception Flows (Unhappy Paths):

- At step 7: Validation errors during submission:
  - Missing fields or invalid values → system highlights issues and returns to step 5.
  - Duplicate username → system prompts user to choose a different one and returns to step 5.

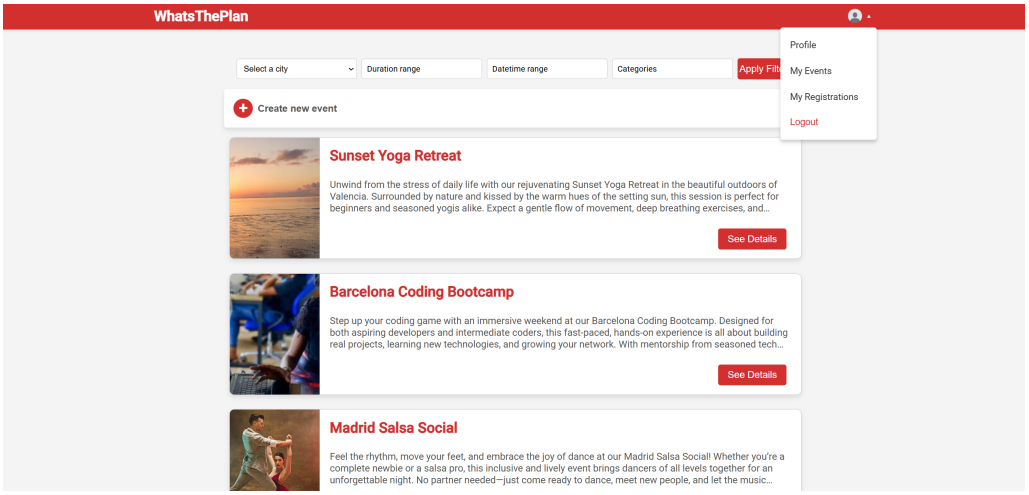


Figure 3.16: Home page with navigation

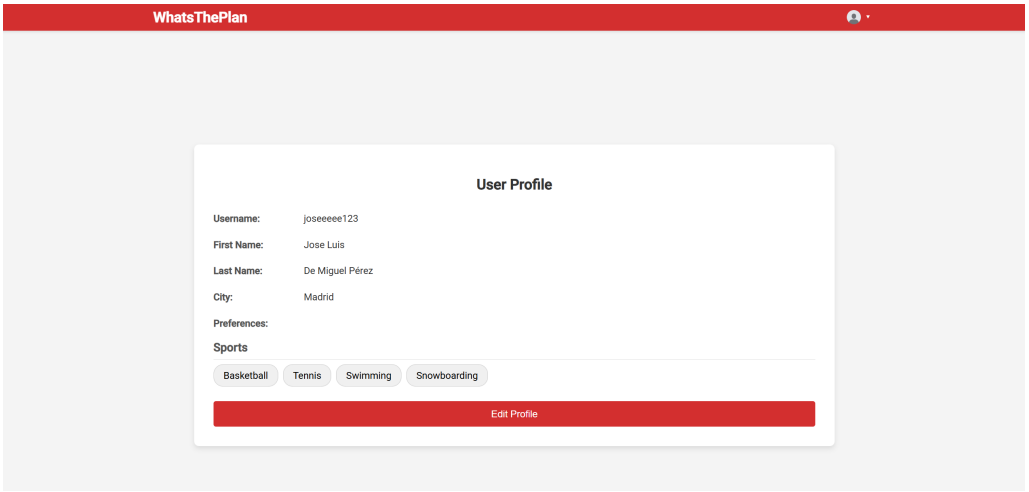


Figure 3.17: Profile details page

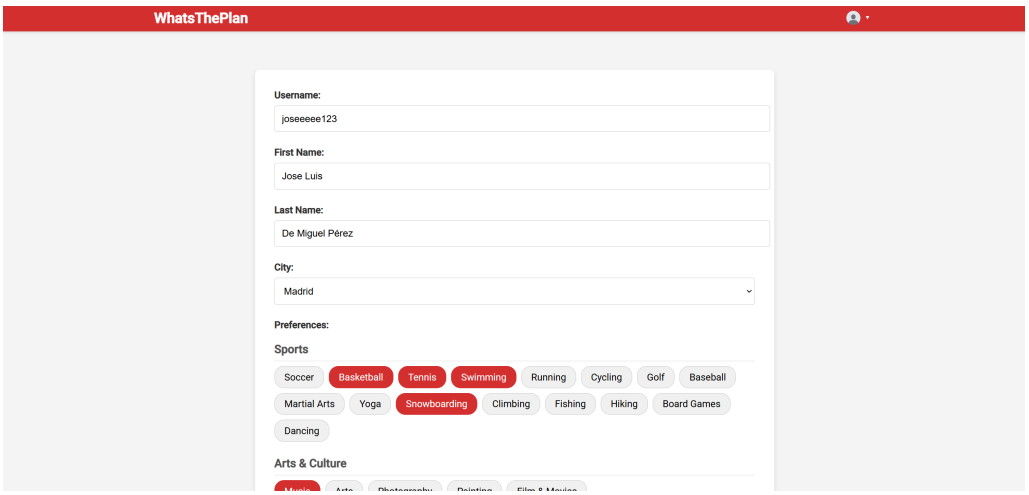


Figure 3.18: Profile update page

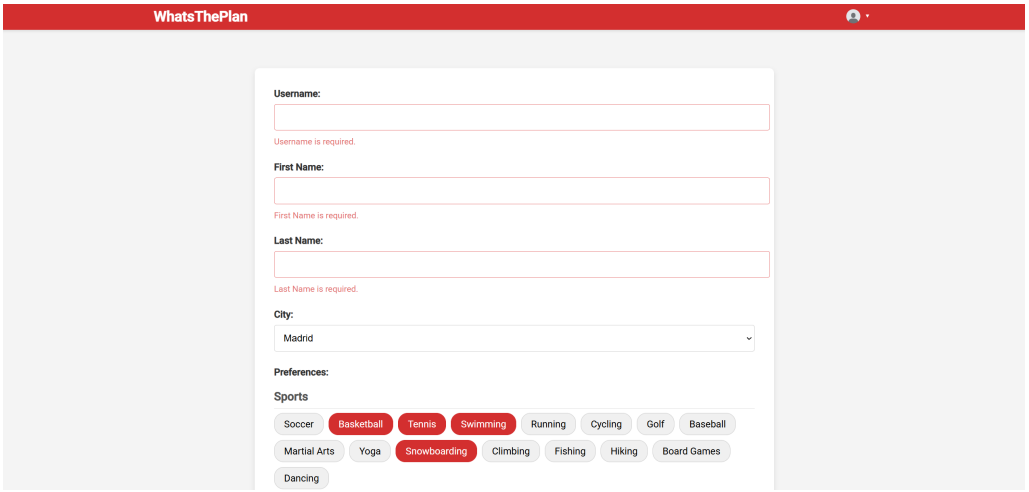


Figure 3.19: Profile update page with invalid input error messages

3.2.4 Event Management

Use Case 4: Event Management

Actor(s): User

Goal: Event Management

Preconditions: User is authenticated and on the Home page.

Postconditions: Event is created, updated or deleted, with cancellation notifications if applicable.

Initial state: User selects “Event Create” from the Home page.

Main Flow (Happy Path):

1. User navigates to the “Event Create” page from Home.
2. System displays the event creation form.
3. User fills in event metadata: title, description, date/time, duration, location, capacity, activity type, image.
4. User submits the form.

5. System validates input data.
6. System stores the event.
7. User is redirected to the Event Details page.
8. User clicks “Event Update”.
9. System displays editable form with current event data.
10. User modifies fields and submits.
11. System validates and saves changes.
12. If needed, user clicks “Delete Event”.
13. System prompts for confirmation.
14. User confirms deletion.
15. System deletes the event and triggers a Cancellation Email.

Alternative Flows:

13b. *User cancels deletion:*

13b1. User declines confirmation prompt.

13b2. System closes dialog and returns to Event Details with no changes.

Exception Flows (Unhappy Paths):

- At step 5: Input validation fails during event creation:
  - Missing or invalid fields (e.g., empty title, past date, invalid duration) → system displays errors and returns to step 3.
- At step 11: Input validation fails during update:
  - Invalid or conflicting updates (e.g., reduced capacity below current attendees) → system shows validation messages and returns to step 9.

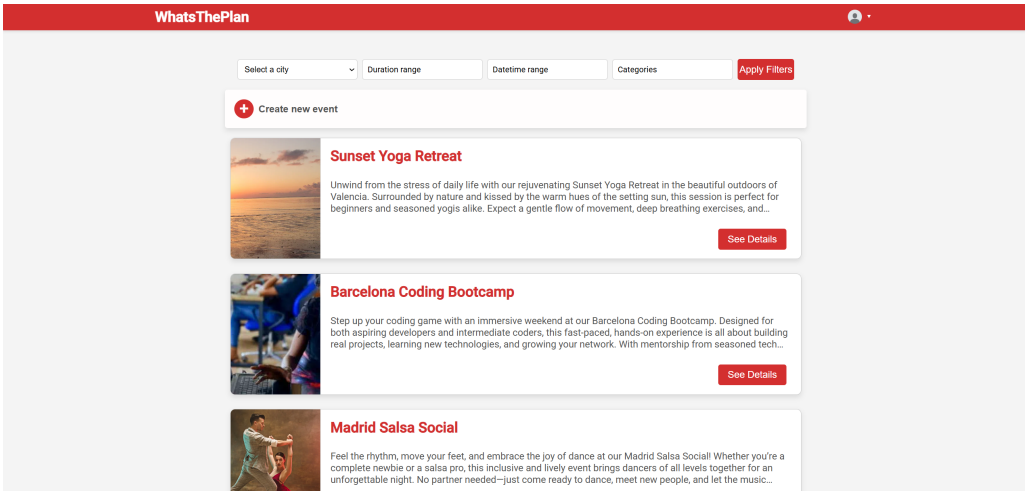


Figure 3.20: Home page

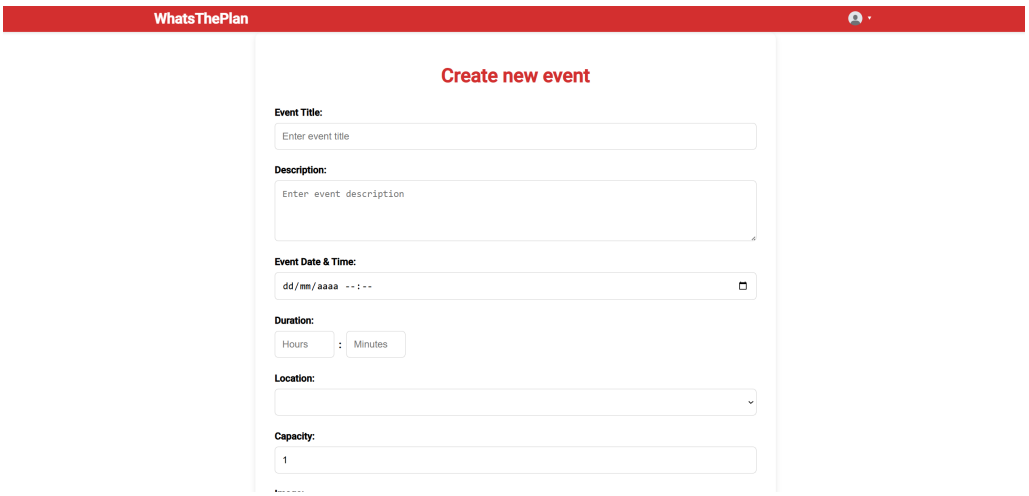


Figure 3.21: Create event page

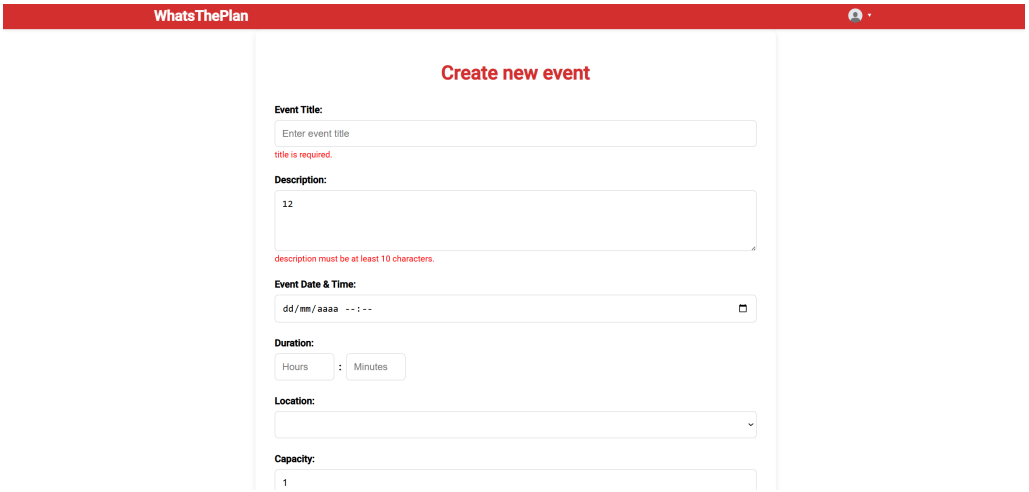


Figure 3.22: Create event page with invalid input error messages

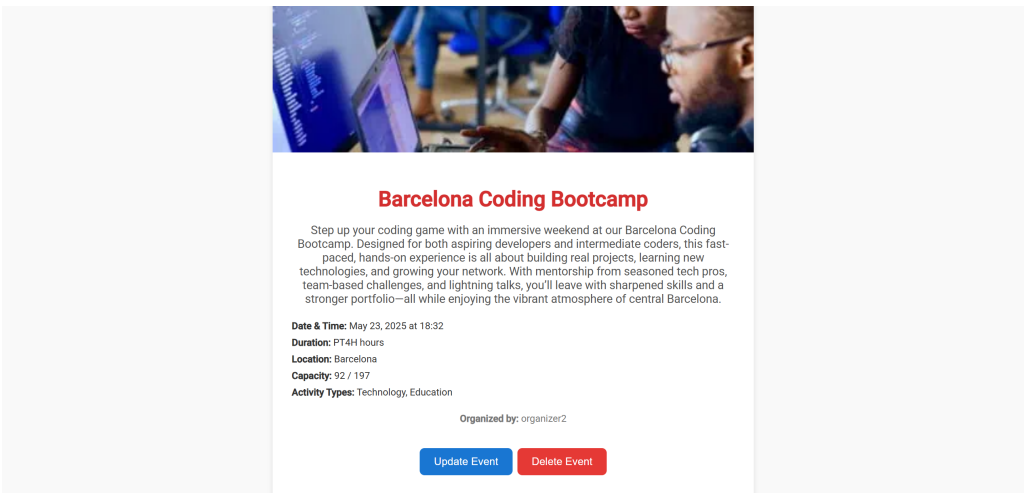


Figure 3.23: Event details page

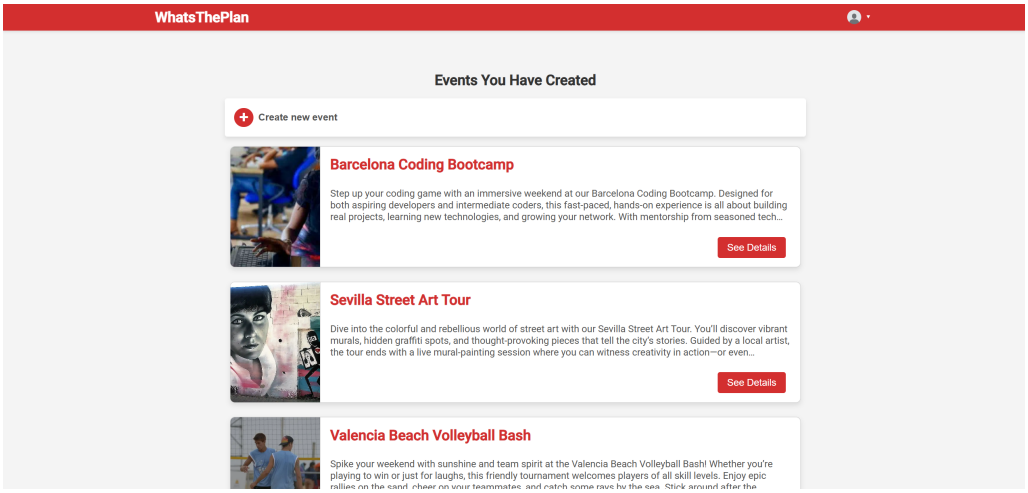


Figure 3.24: My Events page

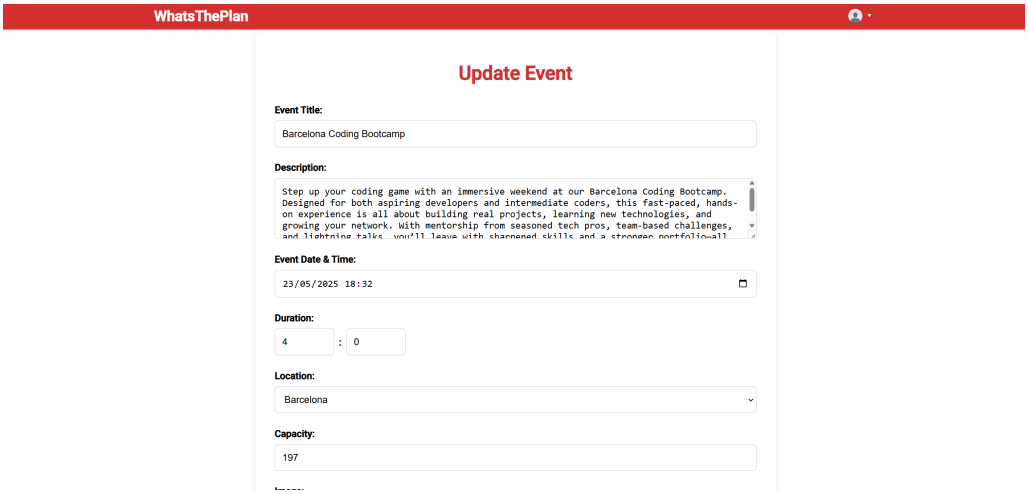


Figure 3.25: Update event page

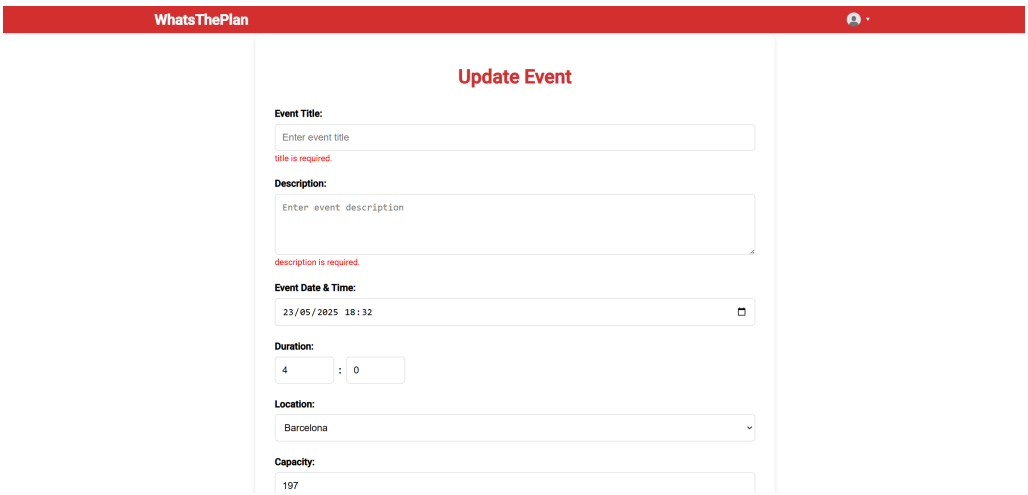


Figure 3.26: Update event page with invalid input error messages

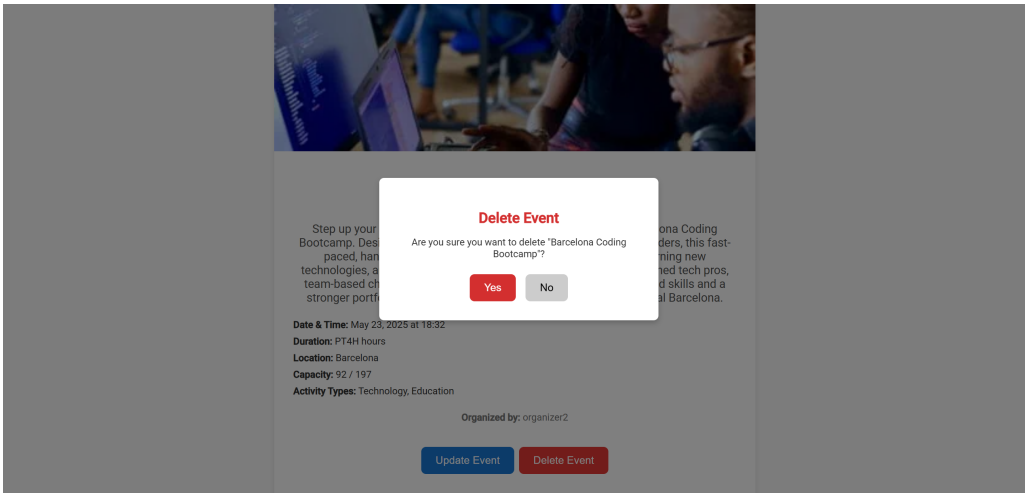


Figure 3.27: Event details page with deletion popup window

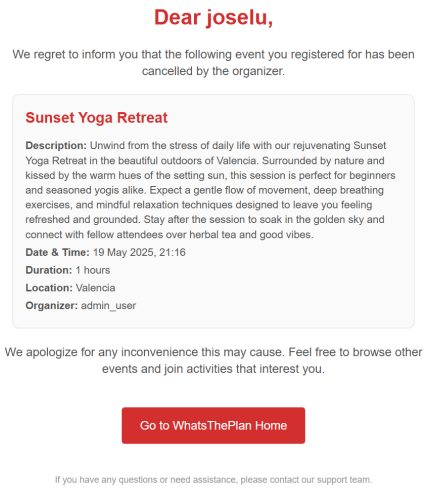


Figure 3.28: Cancellation email registered users receive when event cancellation

3.2.5 Event Discovery and Registration

Use Case 5: Event Discovery and Registration

Actor(s): User

Goal: Event Discovery and Registration

Preconditions: User is authenticated and on the Home page.

Postconditions: User is registered for the selected event and receives a confirmation email.

Initial state: User navigates to the event discovery section from Home.

Main Flow (Happy Path):

1. User accesses the event discovery/search interface in the Home page.
2. User applies optional filters: city, duration range, datetime range, categories.
3. System returns a list of matching events.

4. User selects an event from the list.
5. System displays Event Details.
6. User clicks the “Register Now” button.
7. System checks availability and eligibility.
8. System registers the user for the event.
9. System triggers a Registration Email.
10. Event is now listed in the “My Registrations” section.

Alternative Flows:

3b. *No events match the applied filters:*
- 3b1. System displays a message: “No events found, try other filters”
- 3b2. User is prompted to adjust filters.

6b. *Event is full:*
- 6b1. System disables registration option and displays status message.

6c. *User is already registered for the event:*
- 6c1. System disables registration option and displays status message.

Exception Flows (Unhappy Paths):

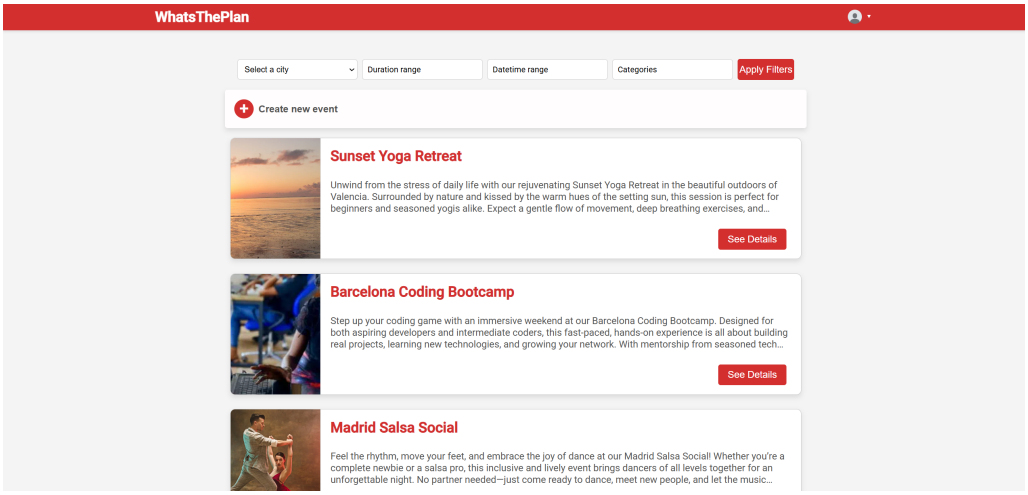


Figure 3.29: Home page

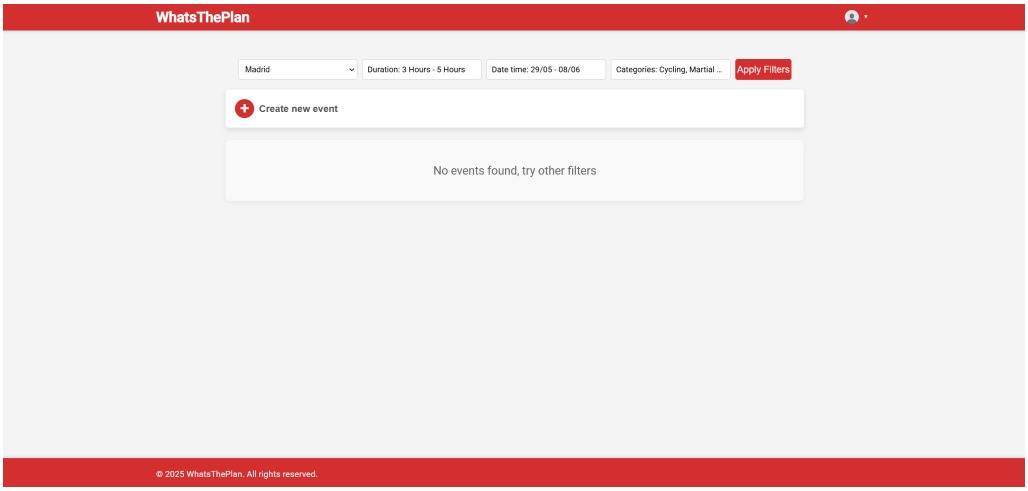


Figure 3.30: Home page with no events matching filters

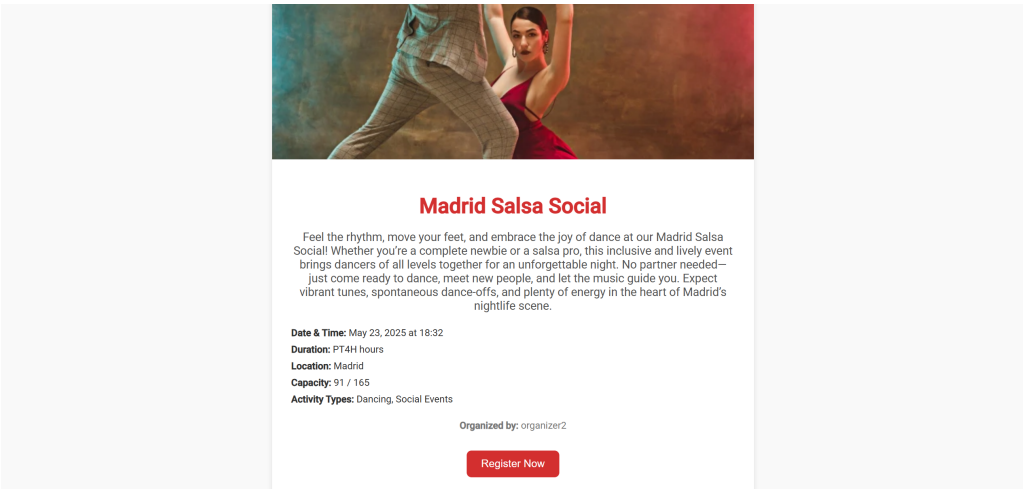


Figure 3.31: Event details page with registration button

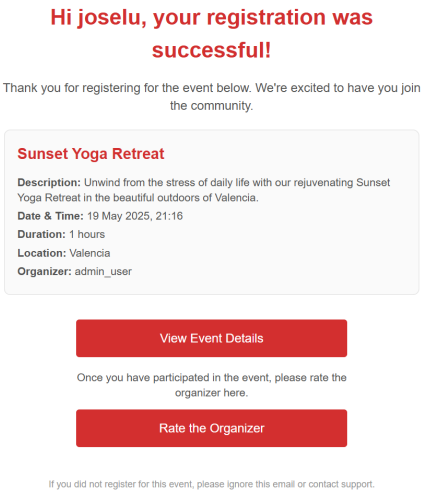


Figure 3.32: Registration email users receive when registration

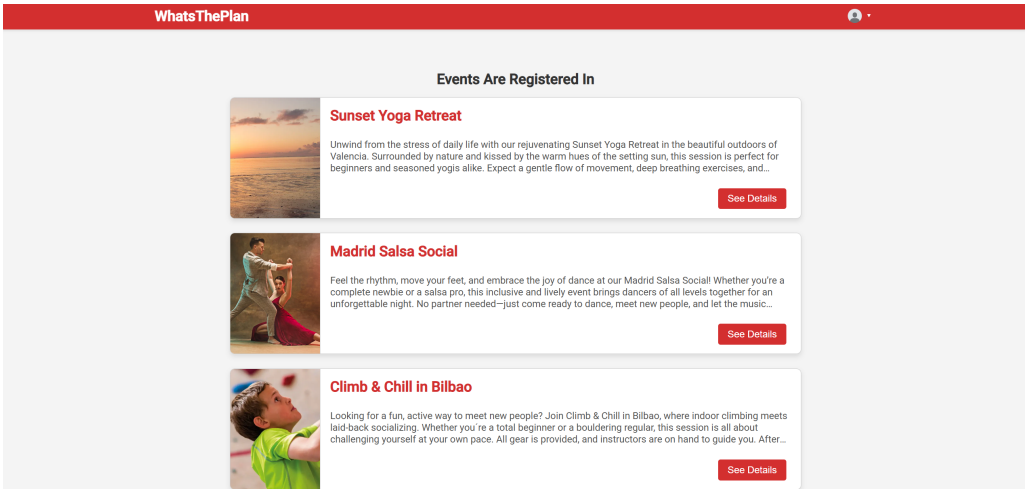


Figure 3.33: My Registrations page

3.2.6 Reviewing Organizers

Use Case 6: Reviewing Organizers

Actor(s): User

Goal: Reviewing Organizers

Preconditions: User is authenticated and has received a Registration Email for an attended event.

Postconditions: User's review is saved and publicly visible on the Event Details page.

Initial state: User clicks the "Submit Review" link from the Registration Email.

Main Flow (Happy Path):

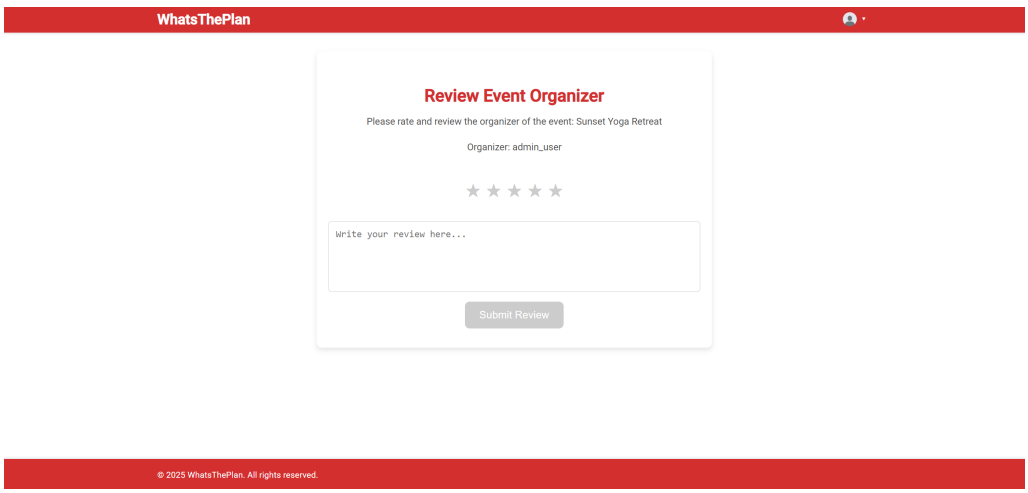
1. User clicks the "Submit Review" link in the Registration Email.
2. System verifies user's participation and opens the Add Review page.
3. User enters a rating (1–5) and optional feedback text.
4. User clicks the "Submit Review" button.
5. System validates the inputs.
6. System stores the review and links it to the organizer.
7. Review becomes visible on the Event Details page under organizer feedback.

Alternative Flows:

Exception Flows (Unhappy Paths):

5b. *Invalid review input:*

5b1. Review text exceeds character limit or is blank when required → system prompts correction.



e

Figure 3.34: Add Review page

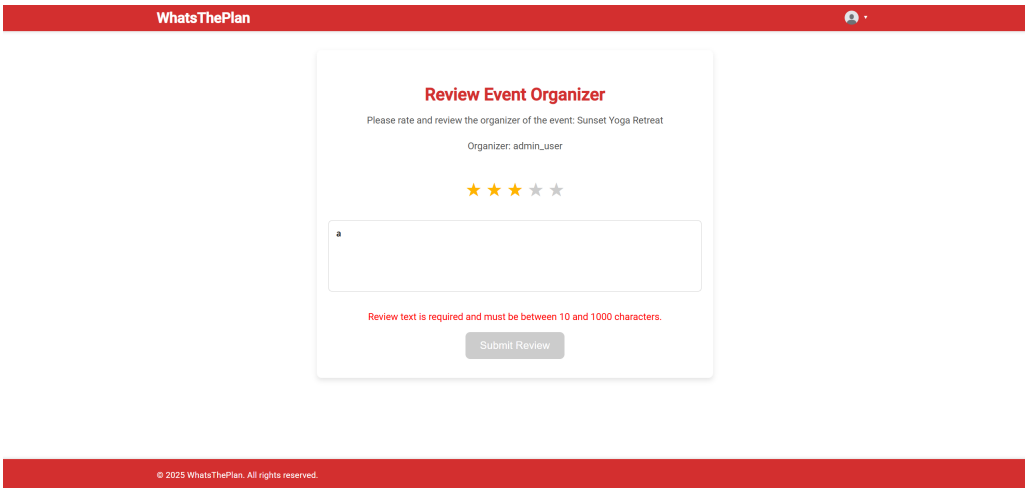


Figure 3.35: Add Review page with invalid input error messages

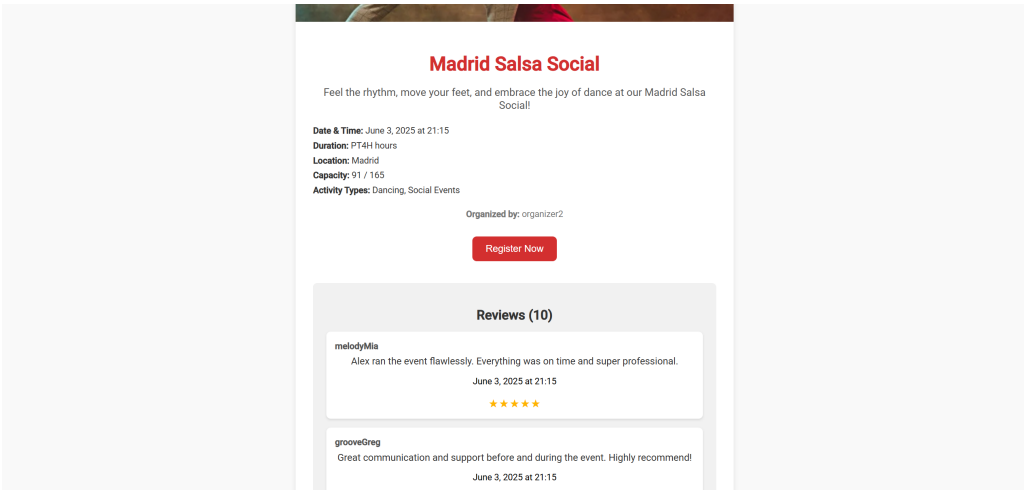


Figure 3.36: Event details page with reviews

4 Software Architecture

4.1 Architectural Decisions

The architectural choices made for *WhatsThePlan* are based on the use cases explained in the previous section. These use cases show the main functions of the platform and point out the different roles inside the system. To handle them well, a modular and scalable design was needed. This makes it easier to change parts of the system separately while keeping everything working together.

This section details the key decisions taken to shape the architecture of the system. It begins by showing the selected architectural style and explaining the reasons of choosing a microservices architecture. Then it outlines how the system is broken down into components based on domain-driven design principles, ensuring clear separation of responsibilities aligned with business domains. Then it is described the data architecture, which focusses on the persistence strategies used in different services. Finally, the section ends with the inter-service communication mechanisms which allow the information sharing between services.

4.1.1 Architectural Style

The *WhatsThePlan* platform combines microservices architecture for the back-end and a front-end to meet the needs of scalability, flexibility, and a high-quality user experience. Microservices enable independent development, scaling, and deployment of services like user management and event handling, while the front-end delivers a dynamic, interactive user interface that communicates with these services via APIs. This dual architecture ensures the platform is adaptable, easy to maintain, and capable of evolving with future needs, balancing both back-end efficiency and an engaging user experience.

4.1.1.1 Microservices Architecture

The *WhatsThePlan* platform uses a microservices architecture. This means the system is built from several small and independent services. Each service handles a specific business task and can be developed, tested, and deployed on its own. These services are not tightly connected and use simple ways to talk to each other, such as HTTP/REST or message queues for asynchronous communication. This design solves problems found in monolithic systems, where even small updates often need the whole system to be redeployed. With microservices, changes can be made and released faster, with less risk and less downtime [12].

The decision to adopt microservices was driven by the diverse functional domains identified during use case analysis (e.g., user management, events, reviews). By mapping these to distinct services, the architecture encourages modular growth and minimizes the complexity of cross-domain coordination.

Some of the key benefits of the microservices architecture are:

- **Scalability:** Each service can be increased or decreased in size on its own, depending on how much work it needs to handle. For example, if more

people use the platform on weekends, the event search service can be expanded without affecting the users or review services.

- **Flexibility in Development:** Development teams can build each service using different programming languages and tools. This lets them choose the most suitable option for each task and encourages faster and more flexible development.
- **Fault Isolation and Resilience:** Because services run separately, a problem in one (like the notification service) does not directly affect the others. This means the system can continue working even when one part fails, improving overall reliability.
- **Ease of Maintenance and Updates:** Services can be updated or deployed on their own. This lowers the chance of breaking other parts and allows small, safe updates more often, which supports continuous delivery.
- **Deployment Speed and Automation:** Microservices align well with CI/CD pipelines. Automated testing, image building, and deployment can be done per service, significantly improving release cycles and reducing time to market.
- **Alignment with Cloud-Native and 12-Factor Principles:** Microservices naturally support the twelve-factor app methodology [13], focussing on statelessness, configuration-driven environments and disposability, all of which improve cloud readiness and operational agility.

Other architecture options, like monolithic and modular monolithic designs, were also looked at during the planning stage. A monolithic style, where everything is built and released as one unit, is simple to start with and easy to test at the beginning. But as the system grows, this approach creates problems. Parts of the system become too connected, updates take longer, and scaling becomes harder and more expensive. The modular monolith improves this a bit by splitting the code into modules, but since it is still deployed as one piece, it keeps many of the same issues.

Because the platform needs to support many different and changing features, and because it must allow parts to be updated or scaled on their own, these two options were not suitable. That is why the microservices architecture was chosen. It fits better with the platform goals for scaling, easy updates and use in the cloud.

To sum up, using microservices gives a strong base for building a system that can grow, is easier to manage, and can adapt to new user needs. It follows good industry practices and supports the platform's long-term plans.

4.1.1.2 Front-end Architecture

The front-end of the *WhatsThePlan* platform is responsible for delivering a dynamic and responsive user interface that interacts seamlessly with the back-end microservices. Built with Angular [4], the front-end presents data from the back-end and manages user interactions for features such as event browsing, profile management, and reviews. It communicates with back-end services via RESTful APIs, ensuring real-time data updates and secure access. The front-end is designed to be modular and scalable, providing a smooth user experience with real-time feedback and input validation.

By separating the front-end from the back-end services, the architecture allows flexibility and ease of maintenance, enabling the platform to evolve with future needs while ensuring an engaging and interactive experience for users.

4.1.2 System Components and Responsibilities

To handle the complexity of the *WhatsThePlan* platform and make sure the software matches the business needs, there has been identified the following components: authentication, fron-end, and back-end microservices.

To determine which microservices was going to have the back-end, it was used the principles of Domain-Driven Design (DDD) [14]. DDD helps design the system based on the main business areas, using clear boundaries called bounded contexts. Each context has its own logic and data. It also encourages teams to use the same language for both developers and business experts. This shared language helps avoid confusion and makes communication and design more accurate.

As explained by Evans, dividing the system into bounded contexts is important to keep the domain logic clear and to allow each part of the system to grow on its own. In *WhatsThePlan*, every bounded context, such as user management, events, reviews, and notifications, is set up as its own microservice. This setup improves modularity and reflects how the platform is organised, both in structure and purpose. These contexts were chosen based on user stories, making sure that the system design fits closely with the features of the platform.

The following sections describe the responsibilities of the key components of the platform.

4.1.2.1 Front-end

Description

The front-end of the *WhatsThePlan* platform is responsible of delivering a dynamic and interactive user interface. It allows users to engage with the platform by interacting with services like event creation, profile management, and browsing through events.

Responsibilities

The front-end presents data provided by back-end services in a user-friendly format. It handles user input, validates it in real-time, and updates the user interface accordingly. The front-end ensures a smooth and responsive experience for the user, displaying information such as events, user profiles, and reviews without requiring full page reloads.

Technology

The front-end is built with Angular due to its framework for developing dynamic, single-page applications (SPAs). Angular offers a modular architecture, making it easier to manage and scale the front-end as the platform grows. It integrates well with back-end services through REST APIs, and secure JWT tokens provided by the Authentication service ensure proper user session management.

Inter-Service Communication

The front-end interacts with the back-end services via RESTful APIs, sending and receiving data related to user actions. It ensures that the user interface remains up-to-date with the latest data and provides a seamless user experience.

4.1.2.2 Authentication service

Description

The Authentication service manages user identity and access on the *WhatsThePlan* platform. It allows users to safely create accounts, log in, and use different parts of the system depending on their role.

Responsibilities

This service takes care of user sign-up, login, and logout. Creates and checks the authentication tokens that are used to confirm who the user is. It also controls what each user can access based on their role. In addition, it handles session safety and account confirmation.

Technology

The service uses AWS Cognito [15], which offers managed user accounts and secure token handling. Using Cognito simplifies the system and follows good security practices.

Inter-Service Communication

This service does not need to talk directly to other services. Instead, it grants access tokens through AWS Cognito. These tokens are checked by each back-end service. This setup keeps authentication in one place, while each service controls access based on the token it receives.

4.1.2.3 User Service

Description

The User Service manages user data that is not part of the login system. It is in charge of saving, updating, and getting user profile details, like usernames, locations, and personal preferences.

Responsibilities

This service handles creating and updating profiles and managing user preferences. It stores information such as usernames, locations, and preferences. It makes sure that only the correct user can change their own data. It also offers APIs, so that other services can access user details when needed.

Technology

This service is built using Java and Spring Boot. This choice helps create secure and easy-to-manage RESTful services.

Inter-Service Communication

Other services use the User Service when they need user-related information. All communication is done using secure REST APIs, and the user identity is checked using tokens included in each request.

4.1.2.4 Event Service

Description

The Event Service manages all event data and actions on the platform. It lets users create, change, and delete events. It also allows users to find events by using filters like category, date, or location.

Responsibilities

This service takes care of adding, updating, and getting event information. It checks the input from users, keeps track of how many people can join each event, and makes sure that only the person who created the event can edit or delete it. It also connects users to the events they join and provides functions for filtering and sorting events for the user interface.

Technology

The service is built in Java using Spring Boot. This setup helps create safe and easy-to-manage RESTful services.

Inter-Service Communication

The Event Service communicates with the User Service to find out who the event organizer is. This is done through secure, synchronous REST API calls, using access tokens to check the identity and permissions of the user making the request.

4.1.2.5 Reviews Service

Description

The Reviews Service is responsible for managing user-submitted feedback on events. It allows participants to rate and review the organisers for the events they have attended.

Responsibilities

This service is in charge of the creation, updating, and retrieval of event organizers reviews. It ensures that only verified participants can provide feedback, validates rating input, and maintains associations between reviews, users, and events. It also calculates and updates aggregated review metrics, such as average rating and total number of reviews, and provides endpoints for accessing these data.

Technology

Implemented in Java with Spring Boot, chosen for its efficiency in building secure, maintainable RESTful services.

Inter-Service Communication

The Reviews Service interacts with User Service to resolve raters and organizer identities. Communication is synchronous and secured via RESTful APIs, with access tokens used to verify the identity and permissions of the requesting user.

4.1.2.6 Notifications Service

Description

The Notifications Service is responsible for delivering system-generated email communications to users in response to key actions or lifecycle events. These notifications enhance user engagement and trust by providing timely updates related to account creation, event participation, and event cancellations.

Responsibilities

This service handles the generation and dispatch of the following types of user-facing emails:

- Welcome emails sent immediately after creating the user account, including a direct link to the platform homepage and account details.
- Email confirmation of registration sent after successful registration of the event, including links to the event details page and the review submission interface.
- Cancellation emails sent to all affected participants when an event is cancelled, providing a clear message and a link to explore other available events.

Technology

Implemented in Java with Spring Boot, the service handles email delivery. Templates are managed through a templating engine (Thymeleaf), enabling dynamic content injection while preserving consistent layout and branding.

Inter-Service Communication

The Notifications Service listens for domain events (user registered, event joined, event cancelled) emitted by the User and Event services via asynchronous messaging through RabbitMQ. This event-driven design allows notification handling to occur independently of the originating business action.

4.1.3 Data Architecture

The data architecture of the *WhatsThePlan* platform follows the idea of decentralized data ownership. Each microservice controls its own database, which helps keep services independent and separate. This setup removes the need for shared databases, avoids tight connections between services, and allows each one to grow, change, and be deployed on its own.

This design matches the ideas of Domain-Driven Design (DDD), where each bounded context includes both its own logic and its own data. Keeping data close to the code that uses it helps set clear limits between services, reduces how much they depend on each other, and makes the system easier to manage. This creates a modular design that is flexible and can handle changes without risking data problems or system stability.

To help with tracking and system checks, all database entries include audit information like when they were created and last updated. This supports the maintainability goal **MR-4**.

The following sections shows all the data storages of each of the microservices.

4.1.3.1 Front-end

The front-end does not store any data itself. Instead, it retrieves all necessary information from the back-end microservices via secure REST API calls. It dynamically displays data, such as user profiles, event details, and reviews, ensuring an interactive user experience.

4.1.3.2 Authentication service

Authentication data is managed entirely by AWS Cognito, which acts as an external identity provider. Instead of storing sensitive credentials within the application infrastructure, emails, passwords, and roles are securely maintained within Cognito user pools. Cognito handles all aspects of authentication, including password hashing and account verification, according to industry-standard security practices.

4.1.3.3 User Service

Data Storage

The User Service uses a PostgreSQL database to store user profile and preference data. PostgreSQL was chosen because it is a relational database, which aligns well with the structured and interconnected nature of the data, such as user accounts and their associated preferences. Its strong support for relational modelling ensures consistency and clarity in managing these relationships.

Relational Schema Overview

The service manages two main tables: `users` and `preferences`. Each user has a unique identifier, username, and email, along with optional fields for name and location. Preferences are modelled as a separate table to allow multiple activity types per user, enabling flexible and normalised storage of user interests.

```
create table users (
    id uuid not null primary key,
    username citext not null unique,
    email varchar(255) not null unique,
    first_name varchar(255),
    last_name varchar(255),
    city varchar(255),
    created_date timestamp default CURRENT_TIMESTAMP,
    last_modified_date timestamp default CURRENT_TIMESTAMP
);
```

Listing 4.1: Users Table

```
create table preferences (
    id uuid not null primary key,
    activity_type varchar(255) not null,
    user_id uuid not null
        references users on delete cascade,
    created_date timestamp default CURRENT_TIMESTAMP,
    last_modified_date timestamp default CURRENT_TIMESTAMP
);
```

Listing 4.2: Preferences Table

4.1.3.4 Event Service

Data Storage

The Event Service uses a PostgreSQL database to persist structured event-related data, including event data, registrations, and category associations. PostgreSQL was selected for managing multi-entity relationships such as many-to-many mappings between events and categories.

Relational Schema Overview

The relational schema consists of four primary tables: `event`, `registration`, `category`, and `event_categories`. Events are core entities, associated with users (organisers), and may belong to multiple categories. Registrations represent the many-to-many relationship between users and events, ensuring uniqueness to prevent duplicate entries. Event-category associations are normalised through a junction table. The schema includes constraints to preserve referential integrity and ensure consistency between deletions.

```
create table event (
    id uuid not null primary key,
    title varchar(255) not null,
    description text,
    date_time timestamp with time zone,
    duration interval,
    location varchar(255),
    capacity integer,
    image_key varchar(255),
    organizer_id uuid,
    created_date timestamp with time zone default now(),
    last_modified_date timestamp with time zone default now(),
    registrations integer default 0
);
```

Listing 4.3: Event Table

```
create table registration (
    id uuid not null primary key,
    event_id uuid not null
        references event on delete cascade,
    user_id uuid not null,
    created_date timestamp with time zone default now(),
    last_modified_date timestamp with time zone default now(),
    constraint unique_event_registration
        unique (event_id, user_id)
);
```

Listing 4.4: Registration Table

```
create table category (
    id uuid not null primary key,
    name varchar(255) not null unique
);

create table event_categories (
    id uuid not null primary key,
```

```
    event_id uuid not null
        references event on delete cascade,
    category_id uuid not null
        references category on delete cascade,
    constraint unique_event_category
        unique (event_id, category_id)
);
```

Listing 4.5: Category and Event-Category Tables

Cache

To improve response times and reduce load on dependent services, the Event Service integrates with Redis as an in-memory cache. Redis is used to temporarily store usernames retrieved from the User Service, and to cache whether an user has registered into an event or not. This avoids repeated synchronous API calls during high traffic operations.

Media storage

All event image are stored in Amazon S3, which provides highly durable and scalable object storage. The PostgreSQL just stores a reference to the S3 object in the column `image_key`. This separation helps the service manage large binary files more easily, without making the database structure too heavy or slowing down the performance of queries.

4.1.3.5 Reviews Service

Data Storage

The Reviews Service store user reviews data in MongoDB. This database was selected because it can easily work with documents that do not follow a fixed structure. It also supports changes in the data format over time. Each saved review contains the rater's identifier and the organiser's identifier. The review also includes the rating value and the date and time it was made. NoSQL databases such as MongoDB makes it easier to save and find review information in a faster way than traditional relational databases.

Document Schema Overview

All reviews are saved in one collection called `reviews`. Each item in this collection has the identifier of the user who wrote the review and the one who received it, a rating as a whole number, an optional text comment, and a timestamp showing when it was created. This way of organising data matches the way MongoDB usually works, where every document is a complete review by itself.

```
{
  "_id": "6fa459ea-ee8a-3ca4-894e-db77e160355e",
  "raterId": "123e4567-e89b-12d3-a456-426614174000",
  "userId": "987f6543-e21c-34d3-b321-654321fedcba",
  "rating": 4,
  "text": "Very well-organized event with great communication.",
  "createdAt": "2024-05-01T14:22:30.001Z"
}
```

Listing 4.6: Sample Review Document

This setup helps quickly find all reviews for a certain user or select them by rating values. Because there are no strict foreign key rules, the system can handle a high number of write operations without problems.

Cache

To make response times faster and reduce the number of requests to the User Service, the Reviews Service uses Redis to store usernames temporarily. When showing reviews, the system first checks Redis for the rater's name. If it is not there, it makes a request to the User service, and then stores the result in Redis for future use. This method makes the system faster, especially when showing many reviews at once, and keeps data correct by using short caching times.

4.1.3.6 Notifications Service

The Notifications service does not use any data storage since it is a stateless service that manages communication without keeping any stored information. Its main task is to send emails to the whole platform.

4.1.4 Inter-Service Communications

The microservices in the platform interact using a combination of synchronous HTTP APIs and asynchronous messaging through RabbitMQ. These endpoints facilitate communication between microservices and also serve as the bridge connecting the front-end to the back-end microservices. The front-end communicates with the back-end services via these RESTful APIs to retrieve and send data.

The following lists summarize the exposed endpoints and message queues for each service, reflecting the communication contracts that enable both service coordination and front-end and back-end interactions.

A complete specification of these APIs is provided in the form of an OpenAPI document, which can be found in Appendix B [Appendix II: API Documentation]. OpenAPI (also known as Swagger) is a widely adopted specification standard for describing RESTful APIs. It allows users to understand the capabilities of a service without direct access to source code or documentation.

User service

- `GET /users` - Retrieve full user profile
- `POST /users` - Create a new user profile
- `PUT /users` - Update existing user profile
- `GET /users-info/{userId}` - Retrieve basic user info (used by other services)

Event service

- `GET /events/{eventId}` - Retrieve detailed event data
- `GET /events` - List events associated with the authenticated user
- `POST /events` - Create a new event (supports multipart/form-data)
- `PUT /events/{id}` - Update event details (supports multipart/form-data)
- `DELETE /events/{eventId}` - Delete an event

- `POST /events/registration/{eventId}` - Register a user to an event
- `GET /events/registration` - Get list of events the user is registered for

Reviews service

- `POST /reviews` - Submit a review for an event organizer
- `GET /reviews/user/{userId}` - Get reviews for a specific user
- `DELETE /reviews/{id}` - Delete a review

Notifications service

- `queue: mail.welcome` - Sends a welcome email when a new user registers
- `queue: mail.registration` - Sends a confirmation email after successful event registration
- `queue: mail.cancellation` - Sends a notification email when an event is cancelled

4.1.5 Software Architecture Overview

This part gives a general overview of the *WhatsThePlan* software architecture. It describes the main parts of the system, which are their functions, and how they work together in different layers. The system is divided into three main layers: presentation, application, and data.

The top is the presentation layer. It includes the front-end, which is what users interact with. The front-end allows users to log in, manage their profiles, handle events, and write or read reviews.

The application layer is where the main back-end services run. These services are built as separate microservices that can be deployed independently. Each takes care of a specific part of the system, such as user profiles, events, reviews, or notifications. The user login and authentication are handled by AWS Cognito, an external service that gives access tokens. These tokens are checked by all other services. The services mostly talk to each other using REST APIs, which is a direct way of communication. For notifications that do not need quick responses, RabbitMQ is used to send messages in the background.

The data layer includes different storage systems selected for each service depending on what it needs. PostgreSQL is used for organised data like users and events. MongoDB stores review data, which can have different formats. Redis helps to make things faster by temporarily storing some data in memory. Amazon S3 is used to store files, such as images, for events.

Figure 4.1 [Software Architecture Overview Diagram] shows this architecture. It presents how the services are connected, the storage each one uses, and the technologies chosen for each part.

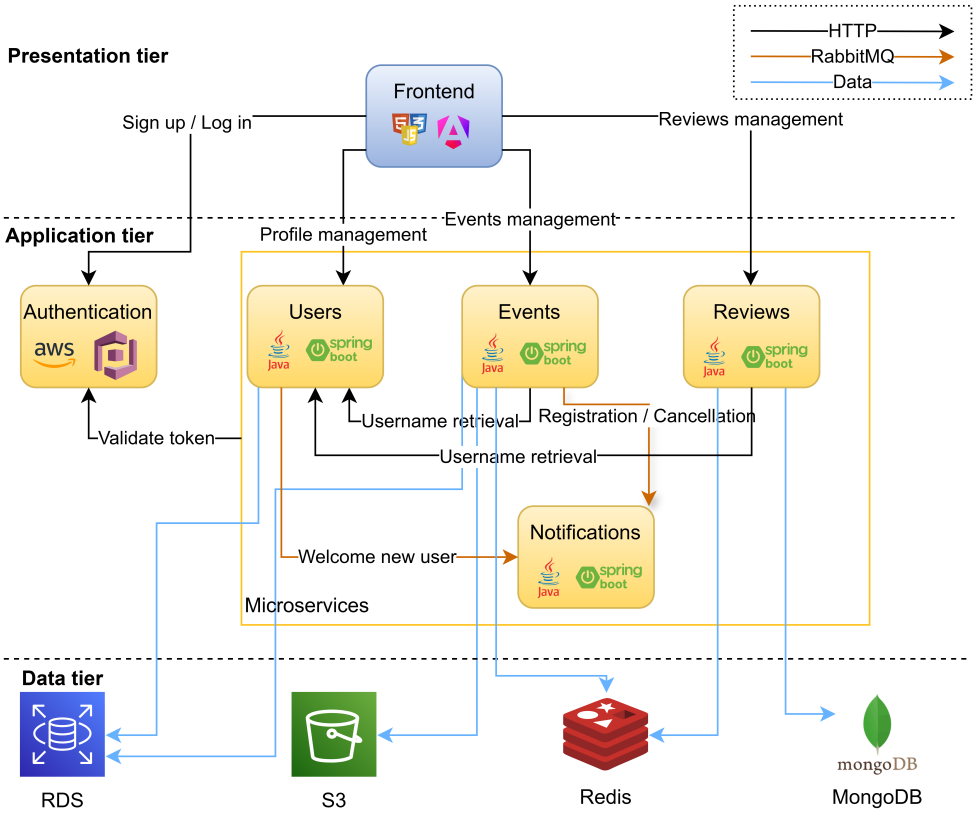


Figure 4.1: Software Architecture Overview Diagram

4.2 Quality Attributes and Architectural Tactics (Software-Level)

This section describes the quality attributes that were used to make decisions about the design and implementation of the *WhatsThePlan* system, and the architectural tactics used to realise them. Each subsection focusses on a specific quality attribute and explains how concrete architectural decisions were made to address the corresponding non-functional requirements. The goal is to ensure that the system not only meets its functional objectives, but also satisfies all non-functional requirements stated in Section 2.2.3 [Requirements].

Some of the architectural tactics used in this section are based on proven strategies from *Software Architecture in Practice* by Bass, Clements, and Kazman [16]. They have been adapted to meet the specific quality goals of the *WhatsThePlan* platform.

4.2.1 Performance Efficiency:

4.2.1.1 Caching

Description

Caching is a performance optimization tactic that stores frequently accessed or computationally expensive data in a fast-access layer, such as an in-memory store, to reduce latency and offload repeated queries from back-end systems. This approach improves overall system responsiveness and supports scalability

under high concurrency.

This tactic corresponds to the "Maintain multiple copies of data" pattern described in *Software Architecture in Practice* by Bass, Clements, and Kazman [16], where duplicate data storage is used to optimize access speed and system efficiency.

Related Requirements

- **PR-1:** The system shall show search results in 2 seconds or less for at least 95% of searches during normal use.
- **PR-2:** The system shall return frequently accessed data (such as event details) within 1 second for at least 95% of requests.

Application in the Architecture

Caching is implemented using Redis as an in-memory data store to reduce latency for frequent data access.

The Event service caches frequent queries such as usernames and user registration status. The Reviews service uses Redis to cache usernames and related metadata from the User service, reducing redundant REST calls.

The cache entries are stored using TTL (Time-To-Live) policies, that removes stored registries from the cache when the time expires, to ensure consistency with the real data. It has been used a TTL of 1 minute. This design reduces dependency on databases during read-heavy operations. Redis also ensures high throughput and sub-millisecond latency under concurrent access conditions.

4.2.1.2 Asynchronous Messaging

Description

Asynchronous messaging is used to move tasks that are not urgent or take a long time into the background. These tasks are handled later, so they do not slow down actions that the user sees. This helps the system respond faster and work better in general.

Related Requirements

- **PR-3:** User-facing operations (registration, event registration, profile updates) shall not be delayed more than 500 milliseconds by background notification tasks.

Application in the Architecture

The system uses RabbitMQ to send messages between services in a way that does not require an instant answer. Tasks that do not need to happen right away, like sending confirmation emails or notifications, are handled separately from the main process.

For example, when a user signs up or joins an event, the related service sends a message to a RabbitMQ queue. The Notifications Service listens to these queues and takes care of sending emails on its own, without affecting the user's main action. This makes sure that the user does not have to wait for emails to be sent. Using a message system like this also helps keep services separate, avoids spreading failures, and makes handling background tasks more efficient.

4.2.1.3 Reactive Programming

Description

Reactive programming leverages asynchronous, non-blocking operations to handle high-throughput data streams with back-pressure mechanisms. It enables the system to scale efficiently under heavy load by minimizing thread usage and latency in I/O-bound scenarios.

Related Requirements

- **PR-4:** The system shall support up to 500 users doing things at the same time (like searching, registering, or viewing) without getting slower (keeping 95% of actions under 2 seconds).

Application in the Architecture

Reactive programming is used in the Event Service, which is the service that handles the most work and gets the most requests. To make sure it can serve many users at the same time without slowing down, the service is built with Spring WebFlux [17].

This reactive stack enables non-blocking request handling and efficient thread management by avoiding the creation of one thread per request. Instead, I/O operations such as database access or inter-service calls are managed asynchronously, freeing up threads to handle additional incoming requests. This allows the Event Service to maintain low response times under high concurrency, even when dealing with operations like filtered event searches or user registration validation.

By adopting reactive programming only where high throughput and scalability are critical, the architecture maintains a balance between complexity and performance. This tactic plays a key role in meeting the system's concurrency requirements while optimising resource utilisation.

4.2.1.4 Pagination

Description

Pagination is a performance optimization tactic that limits the amount of data returned in a single response by dividing large result sets into smaller, navigable subsets (pages). This reduces the computational and I/O load on back-end services, lowers memory usage, and minimises front-end rendering time. Pagination also helps maintain predictable response times even under high data volume or concurrent access.

Related Requirements

- **PR-5:** The system shall use pagination for complicated searches, showing no more than 10 results per page to keep response times fast.

Application in the Architecture

Pagination is used in endpoints that return a lot of data, such as those for finding events. These endpoints use offset-based pagination with query options like `?page=1&size=10` to limit how much data is sent in each request.

The back-end sets a limit of 10 items per page to keep good performance during heavy data operations. On the front-end, the Angular client adds pagination

buttons and loads data step by step. This makes the system feel faster for users and lowers the amount of data sent over the network. With this setup, the system stays stable and easy to use, even when many users are active or the data grows large.

Figure 4.2 [Pagination component in the event discovery interface] shows how the pagination looks in the event search view of the WhatsThePlan platform. The user interface shows a set number of event cards on each page, and users can click to move through more pages of results smoothly.

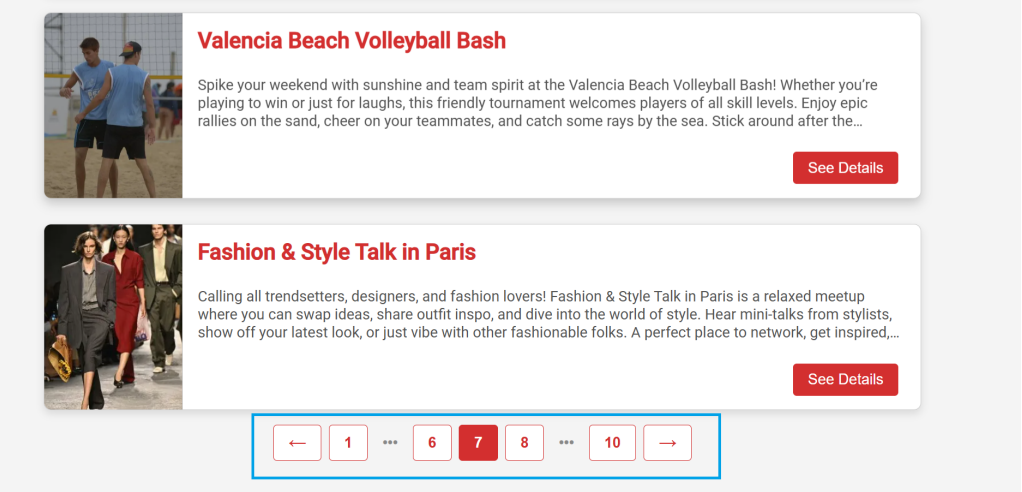


Figure 4.2: Pagination component in the event discovery interface.

4.2.2 Interaction Capability (Usability)

4.2.2.1 User Interface Feedback

Description

User interface feedback is a tactic that provides immediate and clear responses to user actions, such as visual or auditory cues. This enhances the user experience by confirming that inputs have been received and helps maintain an intuitive and seamless interaction flow.

Related Requirements

- **ICR-1**: The system shall give clear and consistent messages after each user action so users can understand what happened (for example, after submitting or selecting something).

Application in the Architecture

The *WhatsThePlan* platform gives quick and clear feedback to users by showing snackbar messages. These messages are made using the `MatSnackBar` tool from Angular Material. Snackbars pop up when a user does something, and they show if the action was successful or if there was a problem. They are easy to see but do not interrupt the user's activity.

This tactic has been applied in the following key user actions:

- Event successfully created.
- Successfully registered to event.

- Event successfully deleted.
- Event successfully updated.
- Review submitted successfully.
- Review deleted successfully.

For example, when a user registers for an event, a snackbar is displayed in the top-right corner of the screen with a confirmation message such as "Successfully registered to event." This feedback confirms the outcome of the action without disrupting the user's workflow.

An example of this feedback mechanism is shown in Figure 4.3 [Snackbar notification after successfully registering to an event].

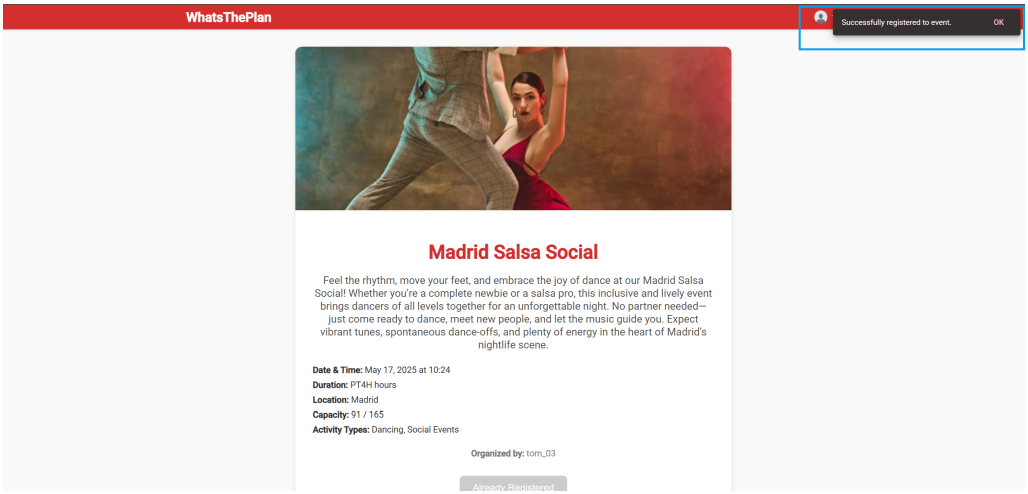


Figure 4.3: Snackbar notification after successfully registering to an event

4.2.2.2 Input Validation and Error Prevention

Description

Input validation and error prevention is a tactic that checks user inputs in real time to ensure they meet defined criteria before processing. This helps prevent invalid or harmful data from entering the system and enhances the user experience by providing clear and immediate feedback on errors.

Related Requirements

- **ICR-2**: The system shall validate user input as they are entered, where applicable, and present immediate, understandable error messages to guide users in correcting invalid data.

Application in the Architecture

User input validation is critical to both the correctness and usability of the *WhatsThePlan* platform. Real-time form validation is implemented using Angular's reactive form module, which ensures that invalid data is caught before being submitted, and that users receive clear, context-specific error messages.

This validation strategy is consistently applied across all forms, including:

- The account confirmation form used after registration to validate email codes.
- The login form to ensure valid credentials are entered.
- The registration form to enforce password complexity, valid email format, and matching confirmation fields.
- The event creation and update forms to validate fields title, date, location, duration, capacity, and selected categories.
- The review submission form to ensure that both rating and optional review text are valid.
- The profile completion form, shown after first login, to collect and validate username, location, and user preferences.
- The profile update form, which applies consistent validation rules for modifying user profile data.

This validation approach not only prevents the submission of invalid data to the back-end services but also significantly improves the user experience by helping users identify and correct mistakes as they occur. Immediate, inline feedback reduces frustration and ensures a smoother interaction flow.

An example of real-time form validation is shown in Figure 4.4 [Input validation in the registration form], which illustrates the registration process when invalid inputs are detected.

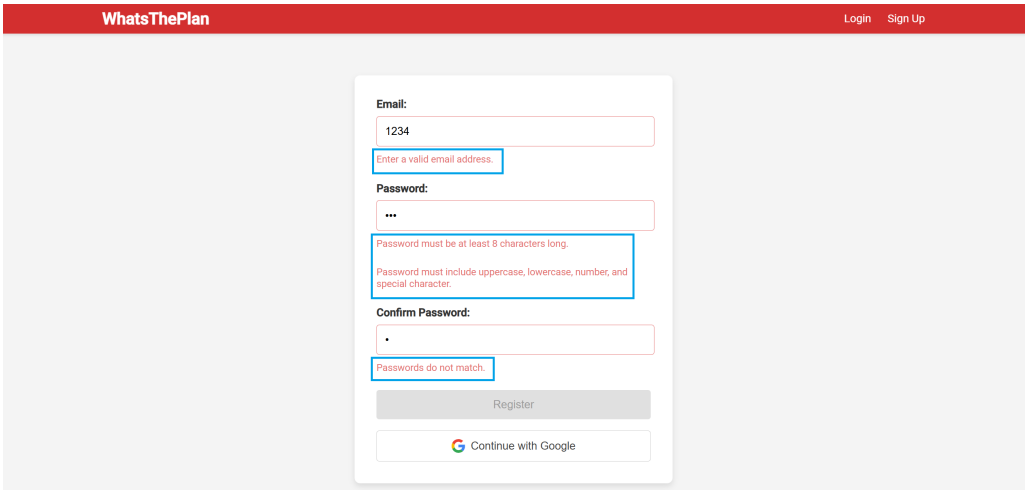


Figure 4.4: Input validation in the registration form

In addition to client-side validation, all forms in the system are protected by server-side validation implemented using Spring Boot's Bean Validation framework. This ensures that even if a user bypasses the front-end checks, any malformed or incomplete request will be rejected before reaching the business logic layer.

Each request payload, such as those for event creation or profile updates, is annotated with validation constraints that enforce the same rules as the front-end. An example is shown in Listing 4.7 [Back-end validation for event creation].

```java
public class EventRequest {
    @NotBlank(message = "Title is required.")
    private String title;

    @NotBlank(message = "Description is required.")
    private String description;

    @NotNull(message = "Date and time must be specified.")
    @Future(message = "Event date must be in the future.")
    private LocalDateTime dateTime;

    @NotNull(message = "Duration is required.")
    private Duration duration;

    @NotBlank(message = "Location is required.")
    private String location;

    @Min(value = 1, message = "Capacity must be at least 1.")
    private int capacity;
}
```

Listing 4.7: Back-end validation for event creation

This layered validation strategy ensures consistency between client and server, improves robustness, and protects the system against malformed or malicious requests.

4.2.2.3 Undo and Cancel

Description

Undo and cancel features let users go back after finishing an action or stop something while it is still happening. This helps avoid mistakes and gives users more trust and comfort when using the system.

These tactics are described in *Software Architecture in Practice* by Bass, Clements, and Kazman [16] as mechanisms that support error recovery and enhance system usability by giving users greater control over their actions.

Related Requirements

- **ICR-3**: The system shall stop users from making important or final changes by mistake by asking for confirmation before doing those actions.

Application in the Architecture

To stop unwanted changes or the loss of data, the *WhatsThePlan* platform asks for user confirmation before doing actions that could have permanent effects. This is done for actions like deleting an event or a review, where a mistake cannot be undone easily.

The system shows a pop-up window (modal dialogue) when the user tries to do such actions. This window explains what the action is and often shows the name of the item being changed, like the event to be deleted. This helps the user check and be sure before continuing.

Only upon explicit confirmation (e.g., clicking “Yes”) does the system proceed with the operation. Otherwise, the dialogue is dismissed and no changes are

made. This design pattern improves user confidence and reduces the likelihood of accidental data loss.

An example of this confirmation mechanism for event deletion is shown in Figure 4.5 [Confirmation dialogue before deleting an event].

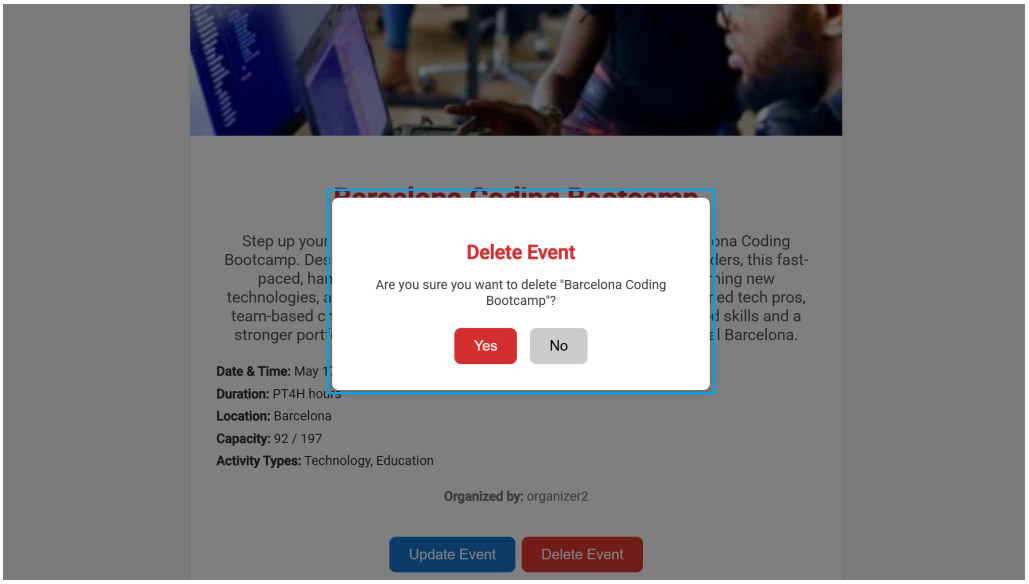


Figure 4.5: Confirmation dialogue before deleting an event

4.2.3 Reliability

4.2.3.1 Retry Pattern

Description

The retry pattern is a tactic that automatically re-attempts failed operations to handle transient faults and temporary outages. This improves system resilience and availability by reducing the impact of intermittent failures and facilitating quicker recovery.

This tactic aligns with the “Retry” pattern described in *Software Architecture in Practice* by Bass, Clements, and Kazman [16], where retrying failed interactions is a recommended strategy for improving fault tolerance in distributed systems.

Related Requirements

- **RR-6**: The system shall keep running and handle requests even if one of its parts fails.
- **RR-9**: The system shall recover normal operation after an interruption within a maximum of 5 minutes.

Application in the Architecture

The retry pattern is used in all HTTP calls between services on the platform to handle short-term problems in a smooth way. These problems can happen when a service is not available for a short time, there are network issues, or the response has a 5xx error (like internal server error or service unavailable).

When one service tries to contact another through an HTTP request, that request is protected by a retry system. If the first try fails, the system tries again

automatically, following a set of retry rules. This is very important in a system made of many services, where small or short-term errors can happen and need to be handled without affecting users.

To prevent putting too much pressure on a service that is down, the retry system uses exponential backoff. This means that the time between each retry grows step by step. It helps reduce traffic and gives the other service more time to fix itself. The backoff settings are adjusted to keep the system quick while still being strong against temporary problems.

- Initial delay: 200 milliseconds
- Multiplier (backoff factor): 2.0
- Maximum delay: 2 seconds
- Maximum attempts: 4

For example, a failed request would be retried after 200ms, then 400ms, then 800ms, and finally 1600ms before ultimately failing. This approach is effective for temporary conditions such as service startup, auto scaling delays, or network congestion.

4.2.4 Security

4.2.4.1 OAuth/OIDC

Description

OAuth 2.0, OpenID Connect (OIDC), and JSON Web Tokens (JWT) are the foundational standards used in modern application security to handle authentication and authorization. These protocols enable secure identity verification and controlled access to protected resources in distributed systems.

- OAuth 2.0 is a system that lets applications get limited access to a user's account on a website without needing to see the user's password [18]. Instead of credentials, it issues access tokens to client applications after successfully authenticating the user or the calling system. These tokens can then be used to authenticate API requests.
- OpenID Connect (OIDC) is a way to confirm a user's identity that works together with OAuth 2.0 [19]. While OAuth defines how access tokens are issued and used, OIDC standardizes how to perform authentication and obtain basic user profile information using an ID token. OIDC allows client applications to verify the identity of the end-user and to obtain information about them.
- JSON Web Tokens (JWTs) are compact, URL-safe tokens that encode claims (i.e., pieces of information about a user or request) and are cryptographically signed [20]. These tokens can be verified by any service using the issuer's public key, making JWTs ideal for stateless, distributed systems.

Related Requirements

- **SR-1**: The system shall prevent unauthorized users from accessing sensitive data and system features.
- **SR-3**: The system shall verify user identity before granting access to sensitive data or restricted actions.

- **SR-5**: The system shall implement session expiration and token renewal to prevent unauthorized account access.

Application in the Architecture

The *WhatsThePlan* platform uses OAuth 2.0 and OIDC for handling login and secure communication. These are set up using AWS Cognito, which supports the main OAuth flows and offers features like user pools, identity connections, and safe token creation.

There are two main ways authentication is done in the system:

- Authorization Code Flow: This is used when users log into the web app. Cognito checks who the user is, either with a username and password or by using another service like Google. After that, it gives two tokens: an ID token to show who the user is and an access token to show what they are allowed to do. These tokens are then added to future HTTP requests to confirm identity and permissions.
- Client Credentials Flow: This is used for communication between back-end services. Each service has its own client ID and secret. It uses these to ask Cognito for a token, which it adds to requests sent to other services. This makes sure that only allowed services can talk to each other.

All tokens are in JWT format. They are signed by Cognito and can be checked by any service using a public key. This lets services confirm who sent the request without storing session data. To keep sessions safe, access tokens expire quickly. If a user is still active, a refresh token can be used to get a new access token without logging in again. This keeps things secure and smooth for the user.

Using OAuth 2.0, OIDC, and JWT is a widely accepted way to secure modern cloud systems [21]. It separates login logic from app code, builds a shared security layer across all parts of the system, and follows the best security practices.

4.2.4.2 Role-Based Access Control (RBAC)

Description

Role-Based Access Control (RBAC) is a security method that controls what users can do in the system based on their role. Instead of giving permissions to each user one by one, RBAC groups permissions under roles. Then, users are given roles based on what they need to do. This makes it easier to manage access and makes sure users only do actions that match their job, which helps keep the system safe and easy to manage.

This method follows the “Authorized actors” idea from the book *Software Architecture in Practice* by Bass, Clements, and Kazman [16], where users are allowed to do certain actions depending on the roles they have. This helps protect different parts of the system by setting clear access limits.

Related Requirements

- **SR-1**: The system shall prevent unauthorized users from accessing sensitive data and system features.
- **SR-2**: The system shall make sure that only users with the correct roles and permissions can do restricted tasks.

Application in the Architecture

In the *WhatsThePlan* platform, RBAC works together with OAuth 2.0 and OpenID Connect (OIDC) by adding user roles inside the JSON Web Token (JWT). When a user logs in using AWS Cognito, they get a signed JWT that includes a `role` field showing their access level. All back-end services can check these roles safely by using Cognito's public keys, which makes sure the role information cannot be changed.

Right now, every user who logs in gets the `user` role. This role is used by the back-end to allow or block actions like creating events, editing profiles, or writing reviews. Each service checks the `role` field in the JWT before letting the user do these actions.

The use of RBAC through token-based claims offers strong extensibility. If future requirements introduce additional roles—such as `admin`—these can be issued by Cognito and encoded directly into the access token. Back-end services can then distinguish between roles and deny or allow access to privileged operations accordingly.

This approach adheres to security best practices for distributed systems, ensuring that access control logic is stateless, scalable, and enforced consistently across all services. It also decouples authentication from authorization, allowing flexible evolution of access policies without modifying the identity layer.

4.2.5 Maintainability

4.2.5.1 Encapsulate

Description

Encapsulation is a method where a service keeps its internal logic and data hidden behind clear and simple interfaces. This means that other parts of the system do not rely on how things work inside the service. As a result, changes inside the service do not affect the rest of the system. This helps make each part easier to manage, test, and update on its own.

This tactic reflects the principle of "Encapsulate" as described in *Software Architecture in Practice* by Bass, Clements, and Kazman [16], where encapsulation is essential to support independent development and evolution of system components.

Related Requirements

- **MR-1**: The system shall be built so that changes in one part do not affect other parts that are not related.
- **MR-6**: Each part of the system must be independently testable without needing the whole system to be running.

Application in the Architecture

Encapsulation is one of the main ideas in the *WhatsThePlan* platform. It is mostly done by using a microservices architecture along with Domain-Driven Design (DDD). Each microservice is built around a specific area of the system. This helps define what it is responsible for and keeps its own logic and data separate from other parts of the system.

This type of architecture supports encapsulation by:

- Stopping other services from directly using its internal databases or business logic.
- Making it possible to create, update, and test each service on its own.
- Letting developers change or replace a service without breaking other parts, as long as the service interface stays the same.

The design also aligns with the SOLID principles, particularly the Single Responsibility Principle (SRP) and the Interface Segregation Principle. By encapsulating behaviour within cohesive services, the system avoids tight coupling and improves long-term maintainability.

Encapsulation is critical for sustaining the system's agility as requirements evolve. It enables to introduce new features, modify existing logic, or optimize internal implementations without introducing regressions in unrelated parts of the application. It also enhances scalability by allowing services to evolve their technology stacks or performance strategies independently.

4.2.5.2 Restrict Dependencies

Description

Restricting dependencies is a method used to control how different parts of the system connect and interact with each other. It helps avoid unwanted links between components, making sure that they communicate only in clear and planned ways. This method helps to make the system more modular, keeps responsibilities separated, and reduces the chance that a change in one part will cause problems in others.

This tactic aligns with the "Restrict dependencies" strategy described in *Software Architecture in Practice* by Bass, Clements, and Kazman [16], where managing dependencies is essential to preserve modularity, support independent evolution, and avoid unexpected effects during system changes.

Related Requirements

- **MR-1**: The system shall be built so that changes in one part do not affect other parts that are not related.
- **MR-2**: Different parts of the system must have clearly separated responsibilities with minimal coupling between them.

Application in the Architecture

In *WhatsThePlan*, the idea of limiting dependencies is used by combining microservices architecture, Domain-Driven Design (DDD), and clear service boundaries. Each service manages its own data and business rules, and other parts of the system can only interact with it through specific HTTP interfaces. There are no shared domain logic libraries or direct database access between services, which helps keep each service's responsibilities separate.

Dependency relationships follow a clear and intentional direction, avoiding circular dependencies and dependency inversions between services. This allows services to evolve independently and reduces the risk of subtle integration bugs caused by hidden assumptions or shared internal states. This also aligns with

the Dependency Inversion Principle from SOLID: services depend on abstractions (HTTP contracts), not on concrete implementations.

Restricting dependencies is especially critical in the context of long-term maintenance and scaling. It allows the architecture to support new features or refactor existing services without introducing architectural debt preserving the system's ability to change with confidence.

4.2.5.3 Database Migrations

Description

Database migrations are a tactic to manage and apply incremental changes to a database schema in a controlled, automated, and repeatable manner. They allow teams to evolve the database structure alongside application code without causing inconsistencies, downtime, or data loss. Migrations ensure that schema changes are versioned, traceable, and reproducible across all environments, from development to production.

Related Requirements

- **MR-3**: It shall be easy to change the database structure without stopping the system or causing errors.

Application in the Architecture

In *WhatsThePlan*, all relational database schema changes are managed using Flyway [22], a widely adopted open-source database migration tool. Flyway applies migration scripts in a specific order, using a versioning system to track which changes have been applied to a given database instance. To keep track of this, Flyway maintains a dedicated table called `flyway_schema_history` in the database, which records details of each applied migration. Each script is tagged with a version number, and Flyway ensures that the migrations are applied consistently and only once.

Flyway runs by itself when the application starts. This means that whenever a service is launched, its database is automatically updated to match the current version of the application. This helps avoid problems like out-of-sync databases or failed deployments caused by missing updates.

This approach is especially important in a microservices architecture, where each service typically manages its own database. Without an automated migration tool, schema changes would cause many errors, and are risky to apply during continuous deployment.

An example from the Event Service illustrates this process:

- `V1__initial_schema.sql`: Defines the base schema with the `event` and `category` tables.
- `V2__create_registration_table.sql`: Introduces a new table to manage many-to-many relationships between users and events.
- `V3__add_registrations_column_in_event.sql`: Adds a new column `registrations` to the `event` table to optimize read performance.

These migrations result in `flyway_schema_history` having the data shown in 4.6 [Flyway `flyway_schema_history` table after migrations].

	version	script	checksum	installed_on	execution_time	success
1	1	V1__initial_schema.sql	219997456	2025-04-12 18:36:46…	593	true
2	2	V2__create_registration_table.sql	-1615408178	2025-04-12 18:36:48…	259	true
3	3	V3__add_registrations_column_in_event.sql	-1371361307	2025-04-12 18:36:49…	143	true

Figure 4.6: Flyway `flyway_schema_history` table after migrations

These migrations are applied in order and tracked by Flyway using the table `flyway_schema_history`. If a migration fails, Flyway halts the process and avoids applying any subsequent scripts, preserving database integrity.

With Flyway, the platform can make changes to the database structure in a way that is easy to control, test, and release. These updates do not cause downtime or create problems in the system. This method is important for supporting continuous delivery and keeping the platform stable as it grows and changes.

4.2.5.4 Executable Assertions

Description

Executable assertions are a software architecture tactic that uses automated tests to verify the correctness of system behaviour and the conformance of individual components to their defined contracts. This tactic enables early defect detection, reduces the risk of regressions, and improves confidence in the system's stability during iterative development and refactoring.

This tactic is discussed as "Executable assertions" in *Software Architecture in Practice* by Bass, Clements, and Kazman [16], where it is recommended as a means to improve system robustness and simplify fault diagnosis during development and maintenance.

Related Requirements

- **MR-6**: Each part of the system must be independently testable without needing the whole system to be running.
- **MR-7**: The system shall support quick checks, using automated tests, to find out if a change has caused any problems.

Application in the Architecture

Automated testing is a very important practice for ensuring the reliability and maintainability of a software system. It allows developers to verify that the behaviour of the system remains correct over time, especially after changes such as feature additions, optimisations, or refactors. Tests are useful in reducing the risk of regressions and enable more confident and agile development.

To correctly test a software system, it is not enough with just testing "happy paths", it should also include edge cases, failure scenarios, and boundary conditions. These tests ensure that the system behaves correctly in all situations.

In *WhatsThePlan*, both unit tests and integration tests are used:

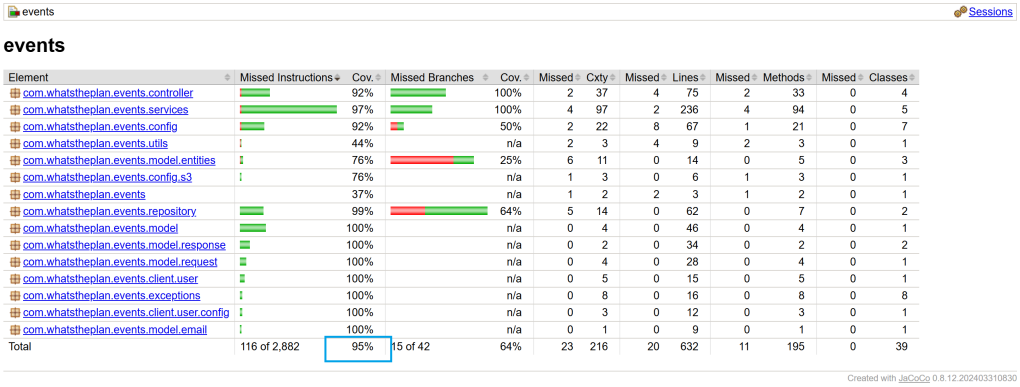
- Unit tests focus on isolated components with mocked dependencies. They are fast, deterministic, and useful for testing core logic and internal edge cases.

- Integration tests verify the interaction between multiple components, including the database, storage systems, or external services. They are essential for validating system-level behaviour and catching issues that only arise through real interactions.

To ensure that the system satisfies functional requirements as expected by users, tests are designed to reflect the acceptance criteria defined in the user stories. These criteria are expressed using the Given-When-Then structure. This structure improves the readability of test cases and ensures traceability between implementation and specification. The following examples from the Event Service illustrate this pattern:

- `givenEventUpdateRequestWithImageUpdate whenRequestIsProcessed thenUploadImageAndStoreEventCategoriesAndImage`
- `givenEventNotOrganizedByUser whenUpdateIsRequested thenReturnForbidden`

To check and improve how much of the code is tested, the system uses JaCoCo (Java Code Coverage) reports. For example, the Event Service now reaches 95% code coverage, as shown in Figure 4.7 [JaCoCo coverage report for the Event Service — 95% instruction coverage]. This high percentage means that most of the code lines and decision points are tested, which helps make the system more reliable and lowers the chance of hidden problems. The complete list of JaCoCo reports for each microservice is shown at C [Appendix III: JaCoCo Reports].


events

Sessions

events

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.whatstheplan.events.controller		92%		100%	2	37	4	75	2	33	0	4
com.whatstheplan.events.services		97%		100%	4	97	2	236	4	94	0	5
com.whatstheplan.events.config		92%		50%	2	22	8	67	1	21	0	7
com.whatstheplan.events.utils		44%		n/a	2	3	4	9	2	3	0	1
com.whatstheplan.events.model.entities		76%		25%	6	11	0	14	0	5	0	3
com.whatstheplan.events.config.s3		76%		n/a	1	3	0	6	1	3	0	1
com.whatstheplan.events		37%		n/a	1	2	2	3	1	2	0	1
com.whatstheplan.events.repository		99%		64%	5	14	0	62	0	7	0	2
com.whatstheplan.events.model		100%		n/a	0	4	0	46	0	4	0	1
com.whatstheplan.events.model.response		100%		n/a	0	2	0	34	0	2	0	2
com.whatstheplan.events.model.request		100%		n/a	0	4	0	28	0	4	0	1
com.whatstheplan.events.client.user		100%		n/a	0	5	0	15	0	5	0	1
com.whatstheplan.events.exceptions		100%		n/a	0	8	0	16	0	8	0	8
com.whatstheplan.events.client.user.config		100%		n/a	0	3	0	12	0	3	0	1
com.whatstheplan.events.model.email		100%		n/a	0	1	0	9	0	1	0	1
Total	116 of 2,882	95%	15 of 42	64%	23	216	20	632	11	195	0	39

Created with JaCoCo 0.8.12.202403310830

Figure 4.7: JaCoCo coverage report for the Event Service — 95% instruction coverage

Automated tests with clear checks (called assertions) are a key part of keeping the platform easy to maintain and high in quality. They help with continuous delivery by making sure that every new version of the system meets both technical rules and real user needs. This keeps the system stable even as it grows and changes.

5 Cloud Architecture

5.1 Infrastructure Components (AWS)

This section describes the main AWS services used to build the platform infrastructure. Each component has been selected to align with cloud native best practices, offering scalability, reliability, and low operational complexity.

5.1.1 Presentation Layer

5.1.1.1 S3 Web Hosting

Amazon S3 [8] is used to store and serve the static web application for the *WhatsThePlan* platform. The front-end of the website, which is built with HTML, CSS, JavaScript (Angular), and other related files, is saved in an S3 bucket set up for static website hosting.

S3 offers strong reliability, can grow easily, and is cost-effective, which makes it a good choice for hosting the user interface of single-page applications. When used together with CloudFront, S3 helps deliver content quickly to users around the world through nearby edge servers.

The hosted website is available to users over HTTPS. This setup reduces the pressure on back-end services, makes deployment easier, and keeps the user interface available and responsive.

5.1.1.2 CloudFront

Amazon CloudFront [23] is a global Content Delivery Network (CDN) that helps deliver static website content from locations closer to the user. This lowers the delay and makes the experience better. It sends files like HTML, CSS, JavaScript, and images from nearby servers to improve speed and access.

In the *WhatsThePlan* system, CloudFront is used to deliver static files stored in S3. It helps pages load faster, supports access from different parts of the world, and ensures that all files are sent using secure HTTPS. When a user sends a request, it is sent to the closest edge location. If the content is already saved there, it is sent right away. If not, CloudFront gets it from S3, saves it, and then sends it to the user.

CloudFront helps the system work better by taking care of static file requests. This reduces the load on the main services and speeds up content delivery in across different regions.

5.1.2 Security Layer

5.1.2.1 Amazon Cognito

Amazon Cognito [15] is a managed service that helps with user login, registration, and access control for both web and mobile apps. It includes features like user pools, identity federation, and token-based login systems (OAuth, OIDC).

In the *WhatsThePlan* platform, Cognito takes care of all tasks related to user access. It handles user registration, log-in, session control, and uses token-based systems to ensure that access is secure.

Using Cognito means that there is no need to create and manage custom login systems. Improves security by following well-known industry rules and grows automatically as more users join. Cognito also stores passwords in a safe way, using standard hashing and salting methods, which meets the platform's security rule **SR-6**.

5.1.3 Application Layer

5.1.3.1 Amazon ECS

Amazon Elastic Container Service (ECS) [24] is a fully managed container orchestration service that runs Docker containers at scale. In *WhatsThePlan*, ECS is used in Fargate mode, which abstracts away server and cluster management, allowing containers to run without provisioning infrastructure.

Each back-end microservice—like users, events, reviews, and notifications—is set up as its own ECS service. These services are described using task definitions and run inside separate containers. This setup helps keep services independent, easier to scale, and better protected if one fails. The container images are created and saved in Amazon Elastic Container Registry (ECR) [25], and ECS uses the latest versions during deployment.

5.1.3.2 Application Load Balancer

The Application Load Balancer (ALB) [26] is a fully managed load balancer in AWS that routes incoming HTTP traffic based on request content. It supports advanced routing features like path-based and host-based routing, making it ideal for microservices and containerized architectures.

In the *WhatsThePlan* platform, the ALB works as the main access point for all back-end services. It takes user requests from the front-end and sends them to the correct ECS service.

The ALB also uses health checks to find and remove any ECS tasks that are not working properly, which helps keep the system running smoothly and without downtime. It works well with ECS by automatically adding or removing containers when services scale up or down. This setup allows the system to handle requests in a flexible and reliable way.

To guarantee secure communication between clients and the platform, the ALB is configured to terminate TLS and enforce HTTPS for all incoming connections, thereby fulfilling security requirement **SR-4**.

5.1.4 Data Layer

5.1.4.1 Amazon RDS

Amazon Relational Database Service (RDS) [27] is a managed service that makes it easier to set up, run and grow relational databases in the cloud. In *WhatsThePlan*, RDS is used with PostgreSQL as the main database to store organised data such as user profiles, events, and registrations.

In this system, microservices such as the User and Event services connect to the PostgreSQL database using secure connections. As the number of users grows, the RDS instance can be scaled to handle more work.

5.1.4.2 MongoDB

MongoDB [6] is a NoSQL database that is flexible, fast, and easy to scale. Instead of using tables like traditional databases, it stores data in documents that look like JSON. This makes it a good choice for data that do not follow a strict format. In *WhatsThePlan*, MongoDB is used in the reviews service, where flexible structure and fast access are important.

Since AWS does not provide MongoDB as a built-in managed service, the solution is to run MongoDB on an Amazon EC2 instance [28]. Amazon EC2 (Elastic Compute Cloud) gives full control over the server, allowing custom software and settings to be used as needed.

5.1.4.3 Redis

Redis [7] is an in-memory data store that works as a fast key-value cache and message broker. It is useful in situations where very quick access to data is needed, like storing data that is often requested or keeping session information.

Redis saves key-value pairs directly in memory, allowing microservices to get data in just a few microseconds. It can be set up with a time-to-live (TTL) to make sure the data stays up to date, and the cache can be cleared or updated when the original data changes.

The system uses Amazon ElastiCache [29] to run Redis. ElastiCache is a managed service from AWS that handles setup, updates, backups, and recovery. It provides fast performance and can handle large amounts of data with low delay.

5.1.4.4 S3 Image Storage

Amazon S3 [8] is used to save uploaded images by users for events in *WhatsThePlan*. This helps move large media files away from the main servers and databases, making the system easier to scale and reducing the complexity of storing large files.

When users create or update an event, they can upload an image that is saved in a special S3 bucket. These images are not sent directly from the back-end, which helps lower bandwidth use and makes responses faster.

5.1.5 Asynchronous Communication Layer

5.1.5.1 RabbitMQ

RabbitMQ [9] is an open-source tool that helps different parts of a system talk to each other without the need for direct connections. It uses a message queue or publish-subscribe model, which means services can send and receive messages through a middle layer. This makes the system easier to scale and protects it from failures in other services.

In *WhatsThePlan*, RabbitMQ is used to handle background tasks and system events, such as sending notifications. This ensures that time-consuming operations do not block real-time user interactions.

The deployment uses Amazon MQ [30], a managed message broker service provided by AWS that supports RabbitMQ natively. Amazon MQ automates maintenance tasks such as provisioning, patching, failover, and monitoring, while providing high availability and secure integration with AWS networking.

5.2 Continuous Integration and Deployment Pipeline

Continuous Integration and Continuous Deployment (CI/CD) is a method in software development that automates the process of building, testing, and deploying code changes. This ensures that every update is built, tested, and ready for deployment with minimal manual intervention. CI/CD accelerates the release process, increases reliability, and helps teams deploy updates more frequently and with greater confidence.

For the *WhatsThePlan* platform, there are two types of CI/CD pipelines, one CI/CD pipeline for the back-end microservices and another for the front-end. The details of each pipeline will be explained in the following sections.

5.2.1 Microservices CI/CD pipelines

5.2.1.1 CI/CD Pipeline

The CI/CD pipeline implemented for *WhatsThePlan* automates the process of building, testing, and deploying application updates. It integrates several AWS-managed services to streamline software delivery across the platform's microservices. There is an individual pipeline for each microservice of the software system.

The process begins when developers push code changes to GitHub [31], a version control platform that hosts the source code. This push event triggers the pipeline execution.

AWS CodePipeline [32] is the core of the pipeline. It is a service that helps to manage the code building, testing, and delivery process by automatically running these steps.

The first stage uses AWS CodeBuild [33], a fully managed build service that compiles source code, runs automated tests to verify the correctness of the code, and generates deployable artifacts. In this case, the artifacts are Docker container images. Section 5.2.1.2 [Build Process] will explain this process with more detail.

These container images are stored in Amazon Elastic Container Registry (ECR), a managed Docker [34] container registry that allows secure storage, versioning, and retrieval of container images.

Finally, AWS CodeDeploy [35], a deployment automation service, retrieves the images from ECR and deploys them to Amazon ECS. During this deployment process, CodeDeploy accesses AWS Systems Manager Parameter Store (SSM) [36] to securely retrieve configuration parameters and secrets required for the environment. Section 5.2.1.3 [Deployment Process] will provide more detail of the deployment process.

The diagram 5.1 [Cloud Architecture Overview for Microservices CI/CD] illustrates the complete architecture of the CI/CD pipeline:

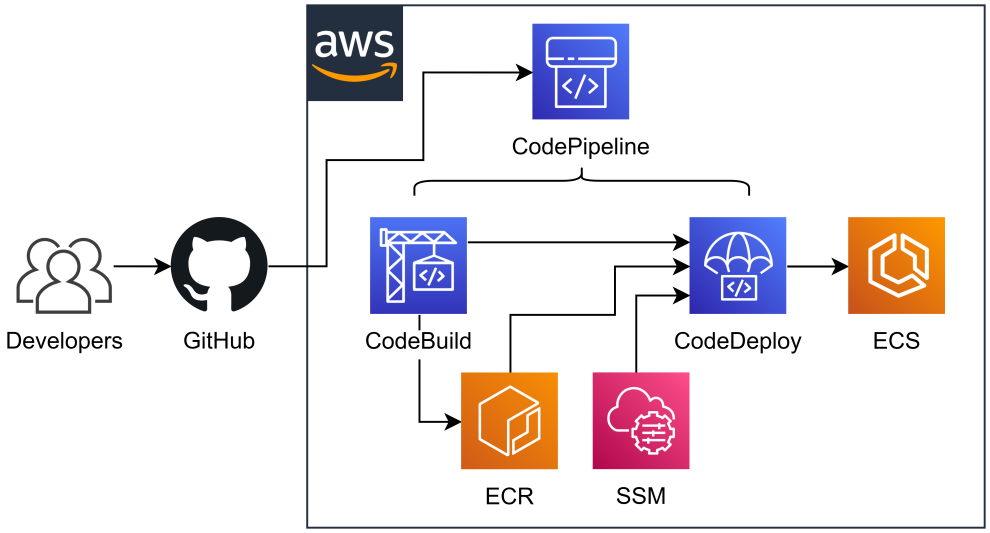


Figure 5.1: Cloud Architecture Overview for Microservices CI/CD

5.2.1.2 Build Process

Once a code change is pushed to GitHub, AWS CodePipeline triggers AWS CodeBuild to initiate the build stage. This process consists of compiling the source code, running unit and integration tests, and building a Docker image of the service. The application is built using Gradle [37], a build automation tool for Java applications. Gradle handles tasks like compiling the Spring Boot microservices, managing dependencies, and running tests. The Dockerfile specifies the instructions to package the application into a container, including compiling the code and preparing a runtime environment. The Dockerfile used for the Java microservices in this project is shown in Section A.1 [Dockerfile].

If all tests pass, the image is marked with a unique tag based on the commit and sent to Amazon ECR, which is the container registry used to store the images for deployment. If any test fails or an error occurs during the build, the pipeline stops immediately, ensuring that only working code moves forward to deployment.

5.2.1.3 Deployment Process

The deployment method used by *WhatsThePlan* combines AWS CodeDeploy with Amazon ECS to update the application using a blue/green deployment model. This method allows for safe updates, avoids downtime, and includes a way to roll back if needed.

In this process, the ECS task definition, specified in a `task-definition.json` file, defines the container image versions, CPU and memory allocations and environment variables for the service tasks. An example of a `task-definition.json` used for the Events Service is shown in Section A.3 [Task definition]. Secrets and configuration values are injected into the containers at startup by referencing parameters stored securely in AWS Systems Manager Parameter Store. This approach ensures sensitive data like credentials are not hardcoded, but dynamically retrieved at runtime. The CI/CD pipeline updates this task definition file with the new container image tags before deploying the updated tasks, ensuring

that the new version is launched correctly with the required settings.

In a blue/green deployment, the current version of the ECS tasks (called the blue environment) keeps handling live traffic. At the same time, a new version (the green environment) is started with the updated code. After checking that everything works, the system slowly moves traffic from the blue environment to the green one.

AWS CodeDeploy supports three traffic shifting options in blue/green deployments:

- All-at-once: Routes 100% of traffic to the new version immediately.
- Canary: Routes a small percentage of traffic first, then shifts the rest after a predefined interval.
- Linear: Shifts traffic in equal increments at regular intervals until 100% of traffic is moved.

The pipeline for *WhatsThePlan* is configured to use the linear strategy, shifting 10% of traffic every 1 minute. This configuration minimizes user impact during deployment and provides an opportunity to detect potential issues early, while still completing the deployment in a timely manner.

During each deployment, ECS health checks keep track of the new tasks. An instance may become unhealthy if it fails to respond to health check requests within a specified time, crashes, or experiences errors that prevent it from functioning correctly. If any of the instances fail or become unhealthy, CodeDeploy will automatically roll back to the last working version. This helps keep the service running without interruption.

With this setup, the platform can deliver updates in a safe and clear way, lowering the risk of errors and increasing trust in the release process.

5.2.2 Front-end CI/CD pipeline

5.2.2.1 CI/CD Pipeline

The CI/CD pipeline for the *WhatsThePlan* front-end automates the process of building, and deploying updates to the user interface. This pipeline integrates several AWS services to ensure that the front-end is built and deployed seamlessly, just like the back-end microservices.

The pipeline process begins when developers push code changes for the front-end to GitHub [31], a version control platform that hosts the source code. This push event triggers the pipeline execution.

At the core of the pipeline is AWS CodePipeline [32], which orchestrates the build, and deployment steps automatically. The first stage uses AWS CodeBuild [33] to build the front-end Angular application. Once the build is completed, the artifacts are prepared for deployment. Sections 5.2.2.2 [Build Process] and 5.2.2.3 [Deployment Process] will explain this process in more detail.

In the deployment phase, the compiled assets are deployed to an Amazon S3 bucket and to CloudFront, ensuring that the latest version of the front-end is available globally. This deployment process ensures that the front-end is always in sync with the most recent code changes.

The diagram 5.2 [Cloud Architecture Overview for Front-end CI/CD] illustrates the architecture of the front-end CI/CD pipeline:

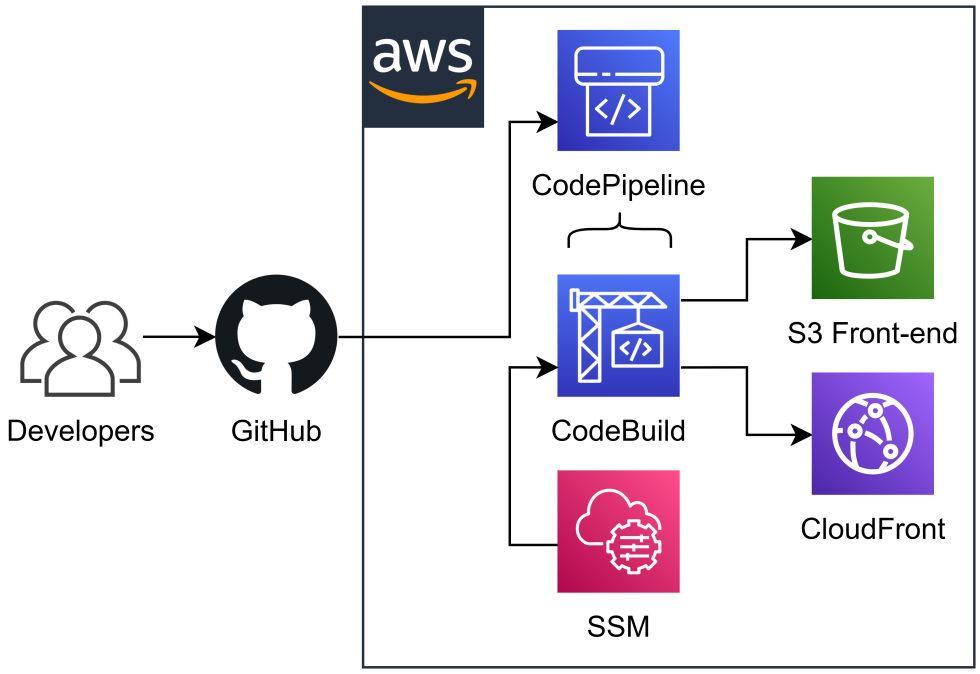


Figure 5.2: Cloud Architecture Overview for Front-end CI/CD

5.2.2.2 Build Process

When a code change is pushed to GitHub, AWS CodePipeline triggers AWS CodeBuild to start the build process.

First, the required dependencies are installed using npm [38], which is a package manager for JavaScript that is widely used to manage dependencies in applications. It automatically installs the libraries and frameworks, such as Angular and other packages, that the front-end requires to run.

Then, the API endpoint is retrieved from AWS SSM Parameter Store, ensuring the front-end uses the correct API during production.

Finally, in the build phase, the Angular application is built, compiling the application and preparing it for production. If any errors occur, the pipeline halts immediately to prevent faulty code from progressing.

5.2.2.3 Deployment Process

The deployment process for the front-end is simpler than for the microservices, as it involves deploying static assets instead of containerized applications. It is directly managed by CodeBuild, after the Angular app is built. The front-end application is built and the resulting static files (HTML, CSS, JS) are deployed to Amazon S3.

Once the build process is complete, the front-end CI/CD pipeline uploads the static files to the appropriate S3 bucket. Since these are static assets, the deployment process does not require complex configuration or service definitions,

making it more straightforward than the back-end microservices deployment.

After the static files are uploaded, Amazon CloudFront is used for content delivery. To ensure users receive the most up-to-date version of the front-end, a CloudFront cache invalidation is triggered. This clears the cached content at CloudFront, forcing the content to be retrieved from the S3 bucket again. This ensures that users always access the latest version of the front-end application without any downtime or manual intervention.

5.3 Cloud Architecture Overview

Figure 5.3 [Cloud Architecture Overview Diagram] shows a general view of the cloud setup used by *WhatsThePlan*. It explains how different AWS services work together to deliver the system to end users.

When a user makes a request, Amazon CloudFront handles it first by serving static files from an S3 bucket where the front-end is stored. Amazon Cognito takes care of user login and session control by creating and checking tokens that allow access to the back-end.

Dynamic requests are routed through the Application Load Balancer (ALB), which directs traffic to microservices deployed in Amazon ECS. Each service runs in an isolated container within the ECS cluster and is independently scalable.

These services connect to several support systems: Amazon RDS for structured data, MongoDB on EC2 for documents, Redis through ElastiCache for caching, and S3 for storing uploaded media. For tasks that happen in the background or need to be delayed, the system uses RabbitMQ, which runs on Amazon MQ.

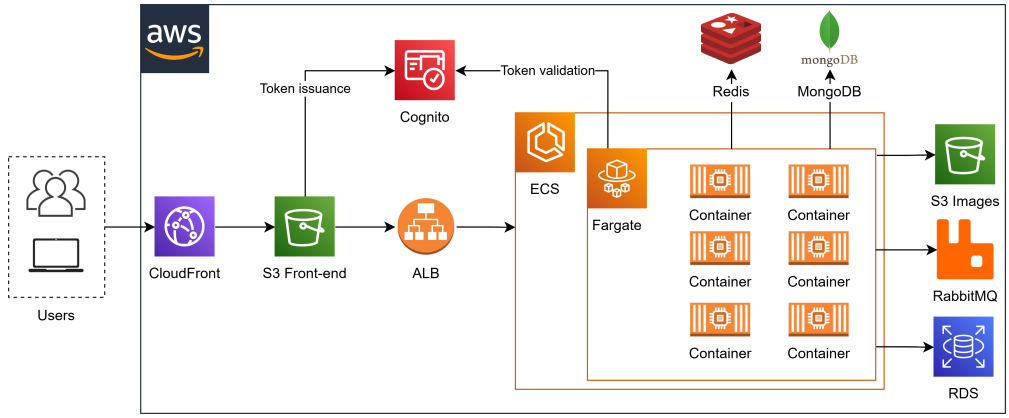


Figure 5.3: Cloud Architecture Overview Diagram

5.4 Quality Attributes and Architectural Tactics (Cloud-Level)

This section focuses on how cloud architectural decisions and deployment strategies contribute to fulfilling the non-functional requirements of the *WhatsThePlan* platform at the infrastructure level.

Some tactics are inspired by patterns from *Software Architecture in Practice* [16], helping to address cloud infrastructure concerns in a production environment.

5.4.1 Reliability

5.4.1.1 Monitor

Description

Monitoring is a method used to collect and study data from different parts of a system by using logs, metrics, and traces. This gives a clear view of how the system is working and helps spot problems early. Good monitoring makes it easier to find the cause of issues and stops users from facing broken or unavailable features.

This idea is described as “Monitor” in the book *Software Architecture in Practice* by Bass, Clements, and Kazman [16], where it is seen as an important part of keeping the system available and making smart decisions during operations.

Related Requirements

- **RR-1**: The system shall perform all specified functions without faults under normal operation, and shall include mechanisms to verify service availability and correct behavior automatically.
- **RR-2**: The system shall have tools like logs, metrics, and traces to help find, understand, and fix problems, and reduce the chance of unnoticed errors.
- **RR-7**: The system shall detect when a component stops working and avoid sending requests to it, using healthy components instead.

Application in the Architecture

Adding logging to the application code is very important to keep track of events and actions across the whole platform. Logs store detailed information on how the system works, including successful operations, errors, and key changes in the state of the system. This information is useful for fixing problems, improving performance, and checking what happened in unexpected situations.

In *WhatsThePlan*, logs generated by microservices are aggregated and managed using AWS CloudWatch Logs [39]. This managed service collects logs centrally from all deployed containers and services, allowing operators and developers to search, filter, and analyse logs in real time.

Figure 5.4 [AWS CloudWatch Logs displaying detailed event service log entries] shows an example CloudWatch Logs console view capturing detailed event service logs, demonstrating how individual requests, cache hits, and database updates are recorded with timestamps and context.

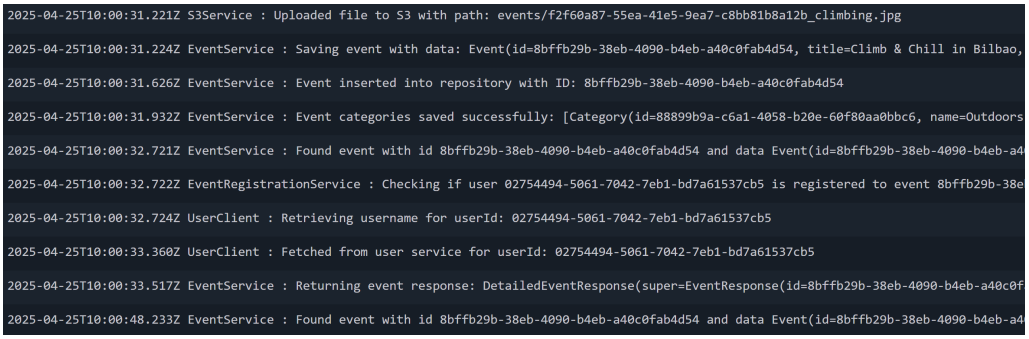


Figure 5.4: AWS CloudWatch Logs displaying detailed event service log entries

By integrating application logs with CloudWatch, the platform achieves enhanced observability, enabling automated alerting of error patterns, and providing the operational insights necessary to maintain system availability and reliability.

5.4.1.2 Heartbeat and Ping

Description

Heartbeat and ping are tactics that involve periodically sending signals or requests to system components to verify their availability and responsiveness. These tactics enable early detection of failures or degraded performance, allowing the system to avoid routing requests to unresponsive or faulty components. This proactive monitoring improves overall reliability and fault tolerance.

This tactic is described as “Heartbeat and ping” in *Software Architecture in Practice* by Bass, Clements, and Kazman [16], where it is recommended as a foundational strategy for maintaining system health and supporting fault-tolerant behaviour.

Related Requirements

- **RR-1**: The system shall perform all specified functions without faults under normal operation, and shall include mechanisms to verify service availability and correct behaviour automatically.
- **RR-7**: The system shall detect when a component stops working and avoid sending requests to it, using healthy components instead.

Application in the Architecture

In the *WhatsThePlan* platform, Spring Boot Actuator **spring-boot-actuator** is used in each microservice to expose health and readiness endpoints that report the service’s status. The health endpoint indicates overall system health by checking critical dependencies such as database connections and message broker availability, while the readiness endpoint signals whether the service is fully initialized and ready to receive traffic. These endpoints allow Amazon ECS and the Application Load Balancer to monitor service health and route requests only to healthy, ready instances, supporting smooth deployments and high availability.

Amazon Elastic Container Service (ECS) and the Application Load Balancer (ALB) collaboratively monitor the health of services by regularly sending HTTP requests to the same Spring Boot Actuator endpoints exposed by each container. ECS

uses these health checks to detect unresponsive instances and automatically triggers recovery actions such as container restarts or task rescheduling, while the ALB routes traffic exclusively to healthy instances. This layered health check strategy improves the resilience of the platform and allows seamless failover.

Using both internal checks from the Spring Boot Actuator and external checks from ECS and ALB, the system can find service issues early. Only healthy containers handle traffic, which keeps the platform stable and available.

By integrating application-level heartbeats with infrastructure-level health checks, the system achieves comprehensive coverage to detect and respond to component failures in real time, which is critical for sustaining the platform's high-reliability objectives.

5.4.1.3 Removal from Service

Description

Removal from service is a tactic that isolates faulty components to prevent them from impacting overall system availability, ensuring that unhealthy instances do not receive traffic, the system can maintain responsiveness and reliability.

This tactic is described as "Removal from service" in *Software Architecture in Practice* by Bass, Clements, and Kazman [16], where it is recommended as a proactive measure to limit the impact of partial failures and improve system fault tolerance.

Related Requirements

- **RR-5**: The system shall maintain service availability by detecting and isolating faulty components, ensuring users are not impacted by individual component failures.
- **RR-7**: The system shall detect when a component stops working and avoid sending requests to it, using healthy components instead.
- **RR-8**: The system shall restart any part that fails automatically, without needing someone to do it manually.

Application in the Architecture

The Removal from Service method is based on the heartbeat and ping features provided by the Spring Boot Actuator health endpoints. These endpoints give regular updates on the health of each microservice instance.

Amazon ECS uses the health signals provided by the Spring Boot Actuator health endpoints to determine the operational status of containers. When ECS detects an unhealthy instance (failing the health checks exposed by these Actuator endpoints) it automatically removes the instance from the service, stops the failing container, and initiates a replacement by launching a new healthy instance. This process happens without manual intervention, enabling rapid recovery and maintaining service availability.

At the same time, the Application Load Balancer (ALB) also checks the health of services using the same Actuator endpoints. ALB makes sure it only sends requests to healthy containers by leaving out the ones that are not working.

By working together, ECS and ALB remove bad instances from the system and replace them with healthy ones. This helps the platform stay reliable and keeps services available to users at all times.

5.4.1.4 Load balancing

Description

Load balancing is a tactic that distributes incoming client requests evenly across multiple service instances or components to prevent bottlenecks and optimize the utilization of available resources. This approach enhances system availability and reliability by ensuring that no single instance becomes overwhelmed and that requests are routed only to healthy components, even during failures or periods of high demand.

Related Requirements

- **RR-3**: The system shall maintain operational status and be accessible to users at least 99.9% of the time.
- **RR-4**: The system shall give users steady and reliable access during normal usage. It shall avoid slowdowns that could make the service unavailable. The expected usage levels will be reviewed regularly as part of system upkeep.
- **RR-6**: The system shall keep running and handle requests even if one of its parts fails.

Application in the Architecture

In the *WhatsThePlan* platform, load balancing is very important to keep the system reliable and able to grow. It ensures that all ECS service tasks share the work fairly. The Application Load Balancer (ALB) handles incoming HTTP requests from users and spreads them across the healthy containers running each microservice.

The ALB uses a round-robin method, which sends each new request to the next available container in a loop. This easy-to-use method helps share the traffic evenly, so no single container gets overloaded or becomes a weak spot.

This load balancing setup supports fault tolerance, scalability, and high availability. By evenly distributing traffic and routing around failed components, the system can maintain consistent performance and user experience even during traffic failures or spikes.

Additionally, load balancing facilitates operational practices such as rolling updates and blue/green deployments, allowing new service versions to be introduced gradually without downtime or disruption to end users.

5.4.2 Maintainability

5.4.2.1 Configuration-time Binding

Description

Configuration-time binding is a tactic that delays the assignment of configuration values to system components until deployment or startup, rather than

hardcoding them at compile time. This enables flexible adjustment of configuration settings without requiring recompilation or rebuilding of artifacts. The tactic supports secure management of configuration by externalizing parameters to dedicated configuration stores or services.

This tactic is described as "Configuration-time binding" in *Software Architecture in Practice* by Bass, Clements, and Kazman [16]. It defers the assignment of configuration values to deployment or startup time, increasing flexibility and reducing change costs compared to compile-time binding. While enabling late binding adds complexity, it is cost-effective by allowing configuration updates without rebuilding or redeploying. This tactic is related to "Interpret Parameters" and "Shared Repositories," which also support dynamic and flexible configuration management.

Related Requirements

- **MR-5**: The system shall allow configuration values to be easily changed or updated without requiring recompilation or redeployment, while ensuring these values are stored and managed securely.

Application in the Architecture

In the *WhatsThePlan* platform, configuration-time binding is implemented by retrieving configuration parameters and secrets from AWS Systems Manager Parameter Store [36] during the deployment and container startup phases. This allows microservices to consume environment-specific values such as database credentials, API endpoints, and feature flags without embedding them in container images or source code.

The integration with the CI/CD pipeline enables dynamic injection of these parameters into the runtime environment, supporting quick changes and secure management through encryption. This practice enhances maintainability by allowing configuration updates independently of application releases and supports the security requirement to protect sensitive data at rest and in transit.

5.4.3 Flexibility

5.4.3.1 Containerization

Description

Containerization is a method that puts an application and everything it needs—like libraries, settings, and other files—into a small, separate unit called a container. These containers run the same way on any system, whether it's a developer's laptop or a cloud server. This helps make sure the app works the same everywhere.

This tactic aligns with the "Package dependencies" strategy described in *Software Architecture in Practice* by Bass, Clements, and Kazman [16], where it is recommended to bundle dependencies with the application to reduce configuration and simplify deployment.

Related Requirements

- **FLR-1**: The product shall be packaged for consistent deployment.

Application in the Architecture

Docker [34] has become the standard technology for containerization due to its efficiency, ecosystem maturity, and ease of use. It works by creating container images that encapsulate an application's code, runtime, system tools, and libraries, sharing the host operating system kernel but isolated at the process level.

In *WhatsThePlan*, Docker is used to package each microservice on its own. This makes it possible to deploy the same container in development, testing, and production without changes. Containers help the app run the same way on any system.

Using containers makes deployment easier because you can build an image once and use it anywhere. This supports stable setups and speeds up the delivery process. It also helps the system grow quickly by allowing new containers to be started as needed.

In the cloud setup, Amazon ECS handles the containers. ECS takes care of the infrastructure, so the focus stays on running and scaling the containers. This allows each microservice to be deployed, updated, and scaled by itself, without affecting the rest of the system.

5.4.3.2 Script Deployment Commands

Description

Scripted deployment commands are a tactic that automate the release process using predefined scripts or pipelines to ensure consistent and repeatable deployments. This reduces manual errors, speeds up delivery, and enables safe rollbacks in case issues arise during deployment.

This tactic corresponds to the "Script deployment commands" strategy described in *Software Architecture in Practice* by Bass, Clements, and Kazman [16], where automation of deployment is emphasized as a key practice for achieving reliable and efficient software delivery.

Related Requirements

- **FLR-4**: The deployment process shall be automated to reduce manual steps and human error.
- **FLR-5**: The system shall allow quick rollback to a previous stable version if a deployment fails or has errors.

Application in the Architecture

The *WhatsThePlan* platform uses AWS CodeBuild, CodePipeline, and CodeDeploy services to implement a fully automated CI/CD pipeline. Central to this automation are the two `buildspec.yml` files, one for the front-end and one for the microservices, each defining the build instructions and deployment commands specific to their respective components.

The `buildspec.yml` file for the microservices explains how the application should be built, tested, and turned into Docker container images. After CodeBuild finishes this process, it sends the ready-to-deploy images — which are stored in Amazon Elastic Container Registry (ECR) — to CodePipeline. Then, CodeDeploy handles the next step by deploying the new version of the app to Amazon ECS, using either a blue/green or rolling update method.

The front-end's `buildspec.yml` handles a similar process for the static assets, where the build commands focus on compiling and bundling the Angular application. The built assets are then deployed to Amazon S3 for global distribution through Amazon CloudFront.

Both `buildspec.yml` used for the microservices and front-end in this project are shown in Section A.2 [Buildspec].

5.4.3.3 Rollback

Description

Rollback using container-based and blue/green deployment is a tactic that enables switching between application versions to quickly revert to a stable state if issues occur. This minimizes downtime and ensures continuity by allowing seamless transitions during updates or rollbacks.

This tactic is aligned with the "Rollback" strategy described in *Software Architecture in Practice* by Bass, Clements, and Kazman [16], where the ability to restore a prior version is considered essential for resilient and fault-tolerant deployment processes.

Related Requirements

- **FLR-5**: The system shall allow quick rollback to a previous stable version if a deployment fails or has errors.
- **FLR-6**: The deployment process shall minimize downtime during updates and rollbacks.

Application in the Architecture

In the *WhatsThePlan* platform, AWS CodeDeploy manages the release of new container versions to Amazon ECS using a blue/green deployment method. While the new containers are being deployed, CodeDeploy checks their health by calling the Spring Boot Actuator endpoints provided by each service.

If CodeDeploy detects that a container is unhealthy — the Actuator endpoint returns an unhealthy status or ECS fails health checks — it automatically aborts the deployment and triggers a rollback. The traffic is redirected back to the previously running stable version, ensuring that users experience no downtime or degraded service. This automated rollback mechanism leverages Actuator's detailed health signals and ECS task status to make real-time decisions about deployment success.

5.4.3.4 Automatic Horizontal Scaling

Description

Automatic horizontal scaling is a tactic that dynamically adjusts the number of running service instances in response to workload changes. By adding instances during traffic spikes and removing them during low demand, this tactic maintains consistent performance, optimizes resource usage, and improves availability by distributing load across multiple replicas.

This tactic combines the "Increase resources" and "Schedule resources" strategies described in *Software Architecture in Practice* by Bass, Clements and Kaz-

man [16], which recommend adaptive allocation of resources and in accordance with demand patterns to maintain service quality under varying loads.

Related Requirements

- **FLR-2**: The system shall automatically use more or fewer resources depending on the current workload, so that performance and availability stay stable without needing someone to make manual changes.
- **FLR-3**: The product shall avoid single points of failure to ensure global availability and stable performance.

Application in the Architecture

Service auto scaling is a feature of Amazon ECS that enables automatic scaling of containerized services based on customizable policies and real-time monitoring data. This feature allows dynamic resource management, allowing the system to automatically adjust the number of running tasks to meet varying workload demands. It has two scaling options: scaling policies and scheduled actions.

Scaling policies define rules that trigger scaling events based on monitored metrics. They should generally remain continuously active to respond immediately to unforeseen workload changes. AWS ECS supports several types of scaling policy that provide flexible control over how and when scaling occurs.

- Target Tracking: Keeps a specific metric (like CPU use, memory use, or the number of ALB requests per target) at a chosen level. If the metric goes too high, ECS adds more tasks. If it drops too low, ECS removes some to save resources.
- Step Scaling: Adjusts task count in increments based on CloudWatch alarm thresholds, enabling precise control for variable workloads. For example, it might add two tasks if active users exceed 1000, five if above 3000, and remove three when below 500.
- Predictive Scaling: Uses historical metric data from CloudWatch to forecast future load and scales the service proactively, increasing capacity ahead of expected spikes to ensure readiness and minimize latency.

Scheduled actions allow changes to be made at set times, such as during busy hours or special events. These actions make sure the system is ready for expected changes in traffic without needing manual updates.

Using these auto scaling methods is important for keeping the system available, fast, and cost-effective as the load changes. By adjusting resources to match real or expected traffic, the platform stays stable and can grow without problems.

6 Results and conclusions

6.1 Overview of Achieved Objectives

This section gives a summary of how well the goals from the introduction 1 [Introduction] have been reached in this Master's Thesis.

- **Requirements Engineering and User Stories:** The project's main goals, both functional and non-functional, as well as user stories, have been fully defined. These details are explained in 2 [Project Definition] and form a strong base for system design and development.
- **Architecture Design:** A modular and scalable system design using microservices and Domain-Driven Design (DDD) has been built and explained in 4 [Software Architecture]. The design focuses on flexibility, concern separation, and easy maintenance, following cloud-native and 12-Factor App principles.
- **Platform Implementation:** The front-end and back-end components of the platform have been implemented according to the defined architectural and functional requirements. Key features such as user management, event organization, and review submission are operational and validated through test scenarios.
- **Cloud Deployment:** The platform has been deployed on Amazon Web Services (AWS) using a production-grade infrastructure. The deployment leverages managed services such as ECS, RDS, ElastiCache, S3, and Cognito to ensure high availability, security, and scalability, as described in 5 [Cloud Architecture].
- **CI/CD Pipeline:** A full Continuous Integration and Continuous Deployment (CI/CD) process has been created using AWS tools such as CodePipeline, CodeBuild, and CodeDeploy. This setup handles testing and deployments automatically, supports blue/green updates, and helps deliver changes quickly and safely.

Overall, all objectives have been fully achieved, contributing to the realization of a cloud-native social networking platform.

6.2 Code Analysis and Implementation Metrics

To understand the amount of work done and how the system is organised, this section includes a numerical analysis of the codebase. The data show how much has been built, how tasks are divided among microservices, and how much testing was done during development.

Table 6.1 [Code Statistics Per Microservice] gives details on the main microservices in the back-end. It shows the number of lines of code (LoC), how many source files there are, how many unit and integration tests were written, and the percentage of code that is covered by tests.

Service	LoC (Java)	Files	Tests	Coverage (%)
User Service	1,851	34	22	88
Events Service	4,387	66	92	95
Reviews Service	1,362	31	23	93
Notifications Service	553	17	7	84

Table 6.1: Code Statistics Per Microservice

- The Events Service stands out as the most complex, with 4,387 lines of Java code across 66 files and the highest number of tests (92). This aligns with its central role in handling event creation, search, registration, and filtering.
- The User Service and Reviews Service are more modest in size, but still exhibit strong test coverage (88% and 93%, respectively), which is essential given their roles in identity data and feedback loops.
- The Notifications Service, while the smallest in size, maintains an adequate test base and achieves 84% coverage, reflecting its auxiliary but critical function in asynchronous communication.

Table 6.2 [Codebase Overview by Component: Number of Files and Lines of Code] aggregates code statistics for both the back-end and the front-end, illustrating the distribution of development effort across the stack.

Component	Files	Lines of Code
Back-end (Java)		
User Service	34	1,851
Events Service	66	4,387
Reviews Service	31	1,362
Notifications Service	17	553
Front-end		
HTML	18	967
TypeScript	40	2,443
CSS	18	2,129
Total (All Components)	**224**	**13,692**

Table 6.2: Codebase Overview by Component: Number of Files and Lines of Code

- The front-end comprises 76 files and 5,539 lines of code, divided into TypeScript for logic (2,443 LoC), HTML for structure (967 LoC), and CSS for styling (2,129 LoC). This indicates a well-layered SPA architecture consistent with modern web application development practices.
- The back-end spans 148 files and 8,153 lines of Java code, emphasizing the modular structure achieved through microservices.
- Overall, the project consists of 224 files and nearly 13,700 lines of code, evidencing the engineering effort across system layers.

6.3 Limitations of the Study

While the goals of the project were met, several limitations must be acknowledged that affected the depth of the study:

- Infrastructure Constraints Due to AWS Free Tier: The entire platform was deployed using services available under the AWS Free Tier. As a result, the system could not be subjected to large-scale load testing, high-throughput simulations, or high-availability configurations beyond minimal provisioning. Compute, storage, and network resources were all constrained, limiting the opportunity to validate the system's behaviour under realistic production-scale conditions.
- Absence of Real-World Users: The system has not been tested by actual end-users. Consequently, no empirical data could be gathered to evaluate the system.

6.4 Future Work

Several areas of improvement and expansion have been identified for future iterations of the platform:

- **Scalability Testing:** Deploying in a budgeted AWS environment would enable load and stress testing to evaluate system performance under realistic traffic.
- **Recommendation Engine:** Implementing personalized event suggestions using user preferences, location, and past activity remains a high-priority enhancement.
- **UI/UX Improvements:** Conducting usability testing with real users can guide refinements in interaction design and accessibility.

6.5 Cost Analysis

This section estimates the monthly operational cost of deploying the *WhatsThePlan* platform on Amazon Web Services (AWS), assuming a realistic production environment. The objective is to quantify infrastructure expenses based on anticipated user traffic and system architecture.

The following parameters define the expected usage profile and infrastructure footprint:

- **Monthly Active Users (MAUs)**: 10,000
- **Peak Concurrent Users**: 200
- **Static Content Delivery**: 10 GB of front-end assets hosted in Amazon S3 and distributed via CloudFront
- **Back-end Architecture**:
  - 4 microservices deployed on Amazon ECS Fargate (users, events, reviews, notifications)
  - Each service auto-scales between 1 and 5 tasks, with an average of 3 concurrent tasks per service

  - Task size: 1 vCPU and 2 GB RAM
- **Database Layer**:
  - Amazon RDS with PostgreSQL for structured data (5 relational tables)
  - MongoDB deployed on EC2 for flexible document storage (1 collection)
- **Messaging**: RabbitMQ deployed via Amazon MQ
- **Caching**: Redis cache via Amazon ElastiCache
- **CI/CD Pipeline**: 10 deployments per month per service and front-end, totaling 50 builds across the platform

The table below summarizes the estimated monthly cost for each AWS component, based on AWS service prices at the time of writing (June 2025).

Table 6.3: Monthly AWS Cost Estimate

Resource Type	**Instance / Usage Parameters**	**Cost (USD)**
ECS	12 tasks, 1 vCPU + 2 GB RAM	$426
Load Balancer	1 ALB, 10M requests/month	$23
RDS (PostgreSQL)	db.t3.medium, 100 GB gp2	$160
EC2 (MongoDB)	t3.medium, 100 GB gp3	$38
ElastiCache (Redis)	cache.t3.small	$50
Amazon MQ (RabbitMQ)	mq.t3.small	$130
S3 Storage	60 GB (static + media)	$7
CloudFront CDN	500 GB data transfer	$42.5
Cognito	10,000 MAUs	$55
CodePipeline	5 pipelines, 10 builds each	$7.50
Total Monthly Cost		**$939**

The total estimated monthly cost to run the platform, based on the given setup, is around $939. The biggest cost comes from Amazon ECS, as the back-end services require a lot of computing power. Other important costs come from managed databases (RDS, EC2), Amazon MQ, and content delivery through CloudFront.

6.6 Conclusions

This Master's Thesis has shown the design, development, and cloud deployment of a social networking platform built with a microservices architecture. All main goals—from defining the requirements to setting up cloud deployment and CI/CD pipelines—have been completed successfully.

The platform shows good modular design, strong test coverage, and follows modern cloud practices. Even with some limits, like basic infrastructure and no real users, the results prove that the system is both technically and architecturally possible.

This work creates a good starting point for future improvements, such as large-scale testing, adding personalized features, and improving the user experience. Overall, it shows how current software engineering methods can be applied to build a working system.

7 Bibliography

[1] ISO/IEC/IEEE, *ISO/IEC/IEEE 29148:2018 - Systems and Software Engineering – Life Cycle Processes – Requirements Engineering*, https://www.iso.org/standard/72089.html, Standard, 2018.

[2] M. Cohn, *User Stories Applied: For Agile Software Development*. Boston, MA: Addison-Wesley Professional, 2004.

[3] Spring Team. "Spring boot documentation." Available at https://spring.io/projects/spring-boot (accessed 2025-05-08), VMware, Inc.

[4] Angular Team. "Angular documentation." Available at https://angular.io/docs (accessed 2025-05-08), Google LLC.

[5] PostgreSQL Global Development Group. "Postgresql documentation." Available at https://www.postgresql.org/docs/ (accessed 2025-05-08), The PostgreSQL Global Development Group.

[6] MongoDB, Inc. "Mongodb documentation." Available at https://www.mongodb.com/docs/ (accessed 2025-05-08), MongoDB, Inc.

[7] Redis Ltd. "Redis documentation." Available at https://redis.io/docs/ (accessed 2025-05-08), Redis Ltd.

[8] Amazon Web Services. "Amazon s3 documentation." Available at https://docs.aws.amazon.com/s3/ (accessed 2025-05-08), Amazon Web Services, Inc.

[9] VMware, Inc. "Rabbitmq documentation." Available at https://www.rabbitmq.com/documentation.html (accessed 2025-05-08), VMware, Inc.

[10] Amazon Web Services. "Amazon web services documentation." Available at https://docs.aws.amazon.com/ (accessed 2025-05-08), Amazon Web Services, Inc.

[11] M. Soares. "Cloud computing stats 2025." Available at https://www.nextwork.org/blog/cloud-computing-stats-2025 (accessed 2025-05-08), Nextwork.

[12] M. Fowler and J. Lewis. "Microservices: A definition of this new architectural term." Available at https://martinfowler.com/articles/microservices.html (accessed 2025-05-08), ThoughtWorks.

[13] A. Wiggins. "The twelve-factor app." Available at https://12factor.net/ (accessed 2025-05-08), Heroku.

[14] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2003, ISBN: 978-0321125217.

[15] Amazon Web Services. "Amazon cognito documentation." Available at https://docs.aws.amazon.com/cognito/ (accessed 2025-05-08), Amazon Web Services, Inc.

[16] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd. Boston: Addison-Wesley, 2012, ISBN: 978-0-321-81573-6.

[17] Spring Team. "Spring webflux documentation." Available at https://docs.spring.io/spring-framework/reference/web/webflux.html (accessed 2025-05-08), VMware, Inc.

[18] D. Hardt, *The OAuth 2.0 Authorization Framework*, https://datatracker.ietf.org/doc/html/rfc6749, RFC 6749, 2012.

[19] OpenID Foundation, *OpenID Connect Core 1.0*, https://openid.net/specs/openid-connect-core-1_0.html, 2014.

[20] M. Jones, J. Bradley, and N. Sakimura, *JSON Web Token (JWT)*, https://datatracker.ietf.org/doc/html/rfc7519, RFC 7519, 2015.
[21] T. Kawasaki. "Jwts in oauth and openid connect." Medium article. [Online]. Available: https://darutk.medium.com/jwts-in-oauth-oidc-19c8029551d5.
[22] Redgate Software, *Flyway by redgate: Version control for your database*, https://flywaydb.org/, Accessed: 2025-05-17, 2025.
[23] Amazon Web Services. "Amazon cloudfront documentation." Available at https://docs.aws.amazon.com/cloudfront/ (accessed 2025-05-08), Amazon Web Services, Inc.
[24] Amazon Web Services. "Amazon ecs documentation." Available at https://docs.aws.amazon.com/ecs/ (accessed 2025-05-08), Amazon Web Services, Inc.
[25] Amazon Web Services. "Amazon ecr documentation." Available at https://docs.aws.amazon.com/ecr/ (accessed 2025-05-08), Amazon Web Services, Inc.
[26] Amazon Web Services. "Elastic load balancing - alb documentation." Available at https://docs.aws.amazon.com/elasticloadbalancing/latest/application/ (accessed 2025-05-08), Amazon Web Services, Inc.
[27] Amazon Web Services. "Amazon rds documentation." Available at https://docs.aws.amazon.com/rds/ (accessed 2025-05-08), Amazon Web Services, Inc.
[28] Amazon Web Services. "Amazon ec2 documentation." Available at https://docs.aws.amazon.com/ec2/ (accessed 2025-05-08), Amazon Web Services, Inc.
[29] Amazon Web Services. "Amazon elasticache documentation." Available at https://docs.aws.amazon.com/elasticache/ (accessed 2025-05-08), Amazon Web Services, Inc.
[30] Amazon Web Services. "Amazon mq documentation." Available at https://docs.aws.amazon.com/amazon-mq/ (accessed 2025-05-08), Amazon Web Services, Inc.
[31] GitHub, Inc. "Github documentation." Available at https://docs.github.com/ (accessed 2025-05-08), GitHub, Inc.
[32] Amazon Web Services. "Aws codepipeline documentation." Available at https://docs.aws.amazon.com/codepipeline/ (accessed 2025-05-08), Amazon Web Services, Inc.
[33] Amazon Web Services. "Aws codebuild documentation." Available at https://docs.aws.amazon.com/codebuild/ (accessed 2025-05-08), Amazon Web Services, Inc.
[34] Docker Inc. "Docker documentation." Available at https://docs.docker.com/ (accessed 2025-05-08), Docker, Inc.
[35] Amazon Web Services. "Aws codedeploy documentation." Available at https://docs.aws.amazon.com/codedeploy/ (accessed 2025-05-08), Amazon Web Services, Inc.
[36] Amazon Web Services, *Aws systems manager parameter store documentation*, https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-parameter-store.html, Accessed: 2025-06-01, 2025.
[37] Gradle Inc., *Gradle build tool*, Version 8.8. Accessed: 2025-06-04, 2024. [Online]. Available: https://gradle.org.

[38] npm, Inc., *Npm - node package manager*, Version 10.5.0. Accessed: 2025-06-04, 2024. [Online]. Available: https://www.npmjs.com.

[39] Amazon Web Services. "Amazon cloudwatch documentation." Available at https://docs.aws.amazon.com/cloudwatch/ (accessed 2025-05-08), Amazon Web Services, Inc.

A Appendix I: Deployment Configuration Files

A.1 Dockerfile

```
FROM public.ecr.aws/amazoncorretto/amazoncorretto:21 as builder

WORKDIR /app

COPY gradlew .
COPY gradle gradle
COPY build.gradle .
COPY settings.gradle .
COPY src src

RUN chmod +x ./gradlew

RUN ./gradlew build -x test --no-daemon

FROM public.ecr.aws/amazoncorretto/amazoncorretto:21

WORKDIR /app

COPY --from=builder /app/build/libs/*.jar app.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "app.jar"]
```

Listing A.1: Dockerfile for java microservices

A.2 Buildspec

```
version: 0.2

phases:
  pre_build:
    commands:
      - echo Logging in to Amazon ECR...
      - aws --version
      - aws ecr-public get-login-password --region $AWS_DEFAULT_REGION
          | docker login --username AWS --password-stdin $ECR_URI
      - chmod +x gradlew
      - COMMIT_HASH=$(echo $CODEBUILD_RESOLVED_SOURCE_VERSION | cut -c
          1-7)
      - IMAGE_TAG=${COMMIT_HASH:=latest}
  build:
    commands:
      - echo Build started on `date`
      - ./gradlew clean build
      - echo Building the Docker image...
      - docker build -t $REPOSITORY_URI:latest .
```

```
      - docker tag $REPOSITORY_URI:latest $REPOSITORY_URI:$IMAGE_TAG
  post_build:
    commands:
      - echo Build completed on `date`
      - echo Pushing the Docker images...
      - docker push $REPOSITORY_URI:latest
      - docker push $REPOSITORY_URI:$IMAGE_TAG
      - echo Writing image definitions file...
      - printf '[{"name":"%s","imageUri":"%s"}]' $CONTAINER_NAME
          $REPOSITORY_URI:$IMAGE_TAG > imagedefinitions.json
artifacts:
  files: imagedefinitions.json
```

Listing A.2: buildspec.yml for Microservices

```
version: 0.2

phases:
  install:
    runtime-versions:
      nodejs: 22
    commands:
      - echo Installing dependencies...
      - npm ci
  pre_build:
    commands:
      - echo Getting API parameter from SSM Parameter Store...
      - export API_VALUE=$(aws ssm get-parameter --name "/whatstheplan/
          alb/dns" --with-decryption --query "Parameter.Value" --output
           text --region $AWS_REGION)
      - echo "Substituting API placeholder in environment.prod.ts..."
      - sed -i "s|<API_PLACEHOLDER>|$API_VALUE|g" src/environments/
          environment.prod.ts
  build:
    commands:
      - echo Building Angular app...
      - npm run build --prod
  post_build:
    commands:
      - echo Deploying to S3 bucket $S3_BUCKET...
      - aws s3 sync ./dist/whatstheplan/browser s3://$S3_BUCKET --
          delete --region $AWS_REGION
      - echo Creating CloudFront invalidation...
      - aws cloudfront create-invalidation --distribution-id
          $CLOUDFRONT_DISTRIBUTION_ID --paths "/*" --region $AWS_REGION

artifacts:
  files:
    - '**/*'
  base-directory: 'dist/whatstheplan/browser'
```

Listing A.3: buildspec.yml for Front-end

A.3 Task definition

```
{
  "family": "whatstheplan-<MICROSERVICE_NAME>",
  "executionRoleArn": "arn:aws:iam::<AWS_ACCOUNT_ID>:role/ecsTaskExecutionRole",
  "networkMode": "awsvpc",
  "containerDefinitions": [
    {
      "name": "whatstheplan-<MICROSERVICE_NAME>",
      "image": "<IMAGE_PLACEHOLDER>",
      "cpu": 0,
      "portMappings": [
        {
          "name": "whatstheplan-<MICROSERVICE_NAME>-8080-tcp",
          "containerPort": 8080,
          "hostPort": 8080,
          "protocol": "tcp",
          "appProtocol": "http"
        }
      ],
      "essential": true,
      "environment": [],
      "mountPoints": [],
      "volumesFrom": [],
      "secrets": [
        {
          "name": "AUTH_SERVER_URL",
          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/whatstheplan/cognito/auth-url"
        },
        {
          "name": "AUTH_CLIENT_ID",
          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/whatstheplan/cognito/client-id"
        },
        {
          "name": "AUTH_CLIENT_SECRET",
          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/whatstheplan/cognito/client-secret"
        },
        {
          "name": "AUTH_TOKEN_URL",
          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/whatstheplan/cognito/token-url"
        },
        {
          "name": "DB_HOST",
          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/whatstheplan/rds/host"
        },
        {
          "name": "DB_PASSWORD",
          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/whatstheplan/rds/password"
        },
        {
          "name": "DB_USER",
          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/whatstheplan/rds/user"
        },
        {
          "name": "S3_ACCESS_KEY",
```

```
54          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/
                whatstheplan/s3/access-key"
55        },
56        {
57          "name": "S3_SECRET_KEY",
58          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/
                whatstheplan/s3/secret-key"
59        },
60        {
61          "name": "S3_BUCKET",
62          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/
                whatstheplan/s3/bucket"
63        },
64        {
65          "name": "S3_REGION",
66          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/
                whatstheplan/s3/region"
67        },
68        {
69          "name": "REDIS_HOST",
70          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/
                whatstheplan/redis/host"
71        },
72        {
73          "name": "REDIS_USERNAME",
74          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/
                whatstheplan/redis/username"
75        },
76        {
77          "name": "REDIS_PASSWORD",
78          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/
                whatstheplan/redis/password"
79        },
80        {
81          "name": "RABBITMQ_HOST",
82          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/
                whatstheplan/rabbitmq/host"
83        },
84        {
85          "name": "RABBITMQ_USERNAME",
86          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/
                whatstheplan/rabbitmq/username"
87        },
88        {
89          "name": "RABBITMQ_PASSWORD",
90          "valueFrom": "arn:aws:ssm:eu-west-1:<AWS_ACCOUNT_ID>:parameter/
                whatstheplan/rabbitmq/password"
91        }
92      ],
93      "logConfiguration": {
94        "logDriver": "awslogs",
95        "options": {
96          "awslogs-group": "/ecs/whatstheplan-<MICROSERVICE_NAME>-td",
97          "awslogs-region": "eu-west-1",
98          "awslogs-stream-prefix": "ecs",
99          "awslogs-create-group": "true",
100         "mode": "non-blocking",
101         "max-buffer-size": "25m"
102       },
103       "secretOptions": []
104     },
105     "systemControls": []
106   }
```

```
107   ],
108   "requiresCompatibilities": [
109     "FARGATE"
110   ],
111   "cpu": "512",
112   "memory": "1024"
113 }
```

Listing A.4: task-definition.json for AWS CodeDeploy

B Appendix II: API Documentation

B.1 Users Service API

Swagger Supported by SMARTBEAR

/v3/api-docs Explore

Users Service API OAS 3.1

/v3/api-docs

Servers

http://localhost:8080 - Generated server url

Authorize

User Profile Operations to manage user profile

GET /users Retrieve the profile of the authenticated user

Returns the profile information of the currently authenticated user.

Parameters Try it out

No parameters

Responses

Code	Description	Links
200	User profile retrieved successfully Media type: */* Controls `Accept` header. **Example Value** | Schema	*No links*

```
{
  "username": "john_doe",
  "email": "john.doe@example.com",
  "firstName": "John",
  "lastName": "Doe",
  "city": "Madrid",
  "preferences": [
```

Code	Description	Links

	<pre>"sports",
	"music"
	]
	}</pre>	

401	Unauthorized - user not authenticated	*No links*

PUT `/users` Update the authenticated user's profile

Updates the user profile with the provided new data.

Parameters | Try it out

No parameters

Request body required | application/json

Example Value | Schema

<pre>{
  "username": "john.doe_99",
  "firstName": "John",
  "lastName": "Doe",
  "city": "Madrid",
  "preferences": [
    "sports",
    "music"
  ]
}</pre>

Responses

Code	Description	Links

200	User profile updated successfully
	Media type: */*
	Controls `Accept` header.
	Example Value | Schema
	<pre>{
	"username": "john_doe",
	"email": "john.doe@example.com",
	"firstName": "John",
	"lastName": "Doe",
	"city": "Madrid",</pre>
		No links

Code	Description	Links

	<pre>{
	  "preferences": [
	  "sports",
	  "music"
	  ]
	}</pre>	

400	Invalid request payload	*No links*
401	Unauthorized - user not authenticated	*No links*

POST `/users` Create a new user profile

Creates a new user profile with the provided information.

Parameters | **Try it out**

No parameters

Request body required | application/json

Example Value | Schema

<pre>{
  "username": "john.doe_99",
  "firstName": "John",
  "lastName": "Doe",
  "city": "Madrid",
  "preferences": [
  "sports",
  "music"
  ]
}</pre>

Responses

Code	Description	Links

201	User profile created successfully
	Media type
	/
	Controls `Accept` header.
	Example Value | Schema
	<pre>{
	  "username": "john_doe",</pre>
		No links

Code	Description	Links

```
  "email": "john.doe@example.com",
  "firstName": "John",
  "lastName": "Doe",
  "city": "Madrid",
  "preferences": [
    "sports",
    "music"
  ]
}
```

400	Invalid request payload	*No links*
401	Unauthorized - user not authenticated	*No links*

Basic User Info

Retrieve basic information about users

GET `/users-info/{userId}` Get basic user information by user ID

Returns basic user details such as username, name, and other public profile info.

Parameters | Try it out

Name	Description
userId * required `string($uuid)` *(path)*	UUID of the user to retrieve userId

Responses

Code	Description	Links
200	User information retrieved successfully	*No links*

Media type

/

Controls `Accept` header.

Example Value | Schema

```
{
  "username": "john_doe",
  "email": "john.doe@example.com"
}
```

Code	Description	Links
400	Invalid user ID supplied	*No links*
404	User not found	*No links*

Schemas

UserProfileRequest Collapse all object

User profile creation request

username* Collapse all string [0, 255] characters matches ^[a-zA-Z0-9._-]{3,15}$

Unique username, 3-15 characters; letters, numbers, dot, underscore, hyphen allowed

Example="john.doe_99"

firstName* Collapse all string [0, 255] characters

User's first name

Example="John"

lastName* Collapse all string [0, 255] characters

User's last name

Example="Doe"

city* Collapse all string [0, 255] characters

City where the user lives

Example="Madrid"

preferences Collapse all array<string> [0, 2147483647] items

List of user preferences or interests

Items string ≥ 1 characters

Example Collapse all array

#0="sports"

#1="music"

UserResponse Collapse all object

Detailed user profile response

username Collapse all string

Username of the user

Example="john_doe"

email Collapse all string

Email address of the user

Example="john.doe@example.com"

firstName Collapse all string

User's first name

Example="John"

lastName Collapse all **string**

User's last name

Example="Doe"

city Collapse all **string**

City where the user lives

Example="Madrid"

preferences Collapse all **array<string>**

List of user's activity type preferences

Items **string**

Example Collapse all **array**

#0="sports"

#1="music"

BasicUserResponse Collapse all **object**

Basic user information response

username Collapse all **string**

Username of the user

Example="john_doe"

email Collapse all **string**

Email address of the user

Example="john.doe@example.com"

B.2 Events Service API

Swagger
Supported by SMARTBEAR

/v3/api-docs

Explore

Events Service API

OAS 3.1

/v3/api-docs

Servers

http://localhost:8080 - Generated server url

Authorize

Event Registration

Register users to events

POST /events/registration/{eventId} Register to an event

Registers the authenticated user to the specified event by its UUID.

Parameters

Try it out

Name	Description
eventId * required `string($uuid)` *(path)*	UUID of the event to register to eventId

Responses

Code	Description	Links
201	Successfully registered to the event	*No links*
400	Invalid registration request	*No links*
404	Event not found	*No links*

Code	Description	Links
409	User already registered or event is full	*No links*

GET `/events/registration` Get events the current user is registered for

Retrieves a list of events that the currently authenticated user has registered for.

Parameters | **Try it out**

No parameters

Responses

Code	Description	Links
200	List of registered events returned successfully Media type application/json Controls `Accept` header. **Example Value** | Schema	*No links*


```
{
  "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e",
  "title": "Summer Festival",
  "description": "A day full of music, food, and fun.",
  "dateTime": "2025-07-15T14:00:00",
  "duration": "PT5H",
  "location": "Central Park",
  "capacity": 500,
  "imageKey": "events/123456/image.png",
  "organizerId": "7fa459ea-ee8a-3ca4-894e-db77e160366f",
  "createdDate": "2025-04-01T10:15:30Z",
  "lastModifiedDate": "2025-04-10T12:00:00Z",
  "activityTypes": [
    "music",
    "food",
    "family"
  ],
  "registrations": 120,
  "isOwnedByUser": true
}
```


401	Unauthorized access - user token missing or invalid	*No links*

Events

Operations related to user-created events

PUT `/events/{id}` Update an existing event

Parameters | Try it out

Name	Description
id * required `string($uuid)` *(path)*	UUID of the event to update id

Request body — multipart/form-data

event * required
`object` — Event metadata

image
`string($binary)` — Optional updated image

Responses

Code	Description	Links
200	Event updated successfully Media type: */* Controls `Accept` header. **Example Value** | Schema	*No links*


```
{
  "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e",
  "title": "Summer Festival",
  "description": "A day full of music, food, and fun.",
  "dateTime": "2025-07-15T14:00:00",
  "duration": "PT5H",
  "location": "Central Park",
  "capacity": 500,
  "imageKey": "events/123456/image.png",
  "organizerId": "7fa459ea-ee8a-3ca4-894e-db77e160366f",
  "createdDate": "2025-04-01T10:15:30Z",
  "lastModifiedDate": "2025-04-10T12:00:00Z",
  "activityTypes": [
    "music",
    "food",
    "family"
  ],
```

Code	Description	Links


```
  "registrations": 120,
  "isOwnedByUser": true
}
```


400 Validation error — *No links*

Media type

/

Example Value | Schema


```
{
  "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e",
  "title": "Summer Festival",
  "description": "A day full of music, food, and fun.",
  "dateTime": "2025-07-15T14:00:00",
  "duration": "PT5H",
  "location": "Central Park",
  "capacity": 500,
  "imageKey": "events/123456/image.png",
  "organizerId": "7fa459ea-ee8a-3ca4-894e-db77e160366f",
  "createdDate": "2025-04-01T10:15:30Z",
  "lastModifiedDate": "2025-04-10T12:00:00Z",
  "activityTypes": [
    "music",
    "food",
    "family"
  ],
  "registrations": 120,
  "isOwnedByUser": true
}
```


404 Event not found — *No links*

Media type

/

Example Value | Schema


```
{
  "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e",
  "title": "Summer Festival",
  "description": "A day full of music, food, and fun.",
  "dateTime": "2025-07-15T14:00:00",
  "duration": "PT5H",
  "location": "Central Park",
  "capacity": 500,
  "imageKey": "events/123456/image.png",
  "organizerId": "7fa459ea-ee8a-3ca4-894e-db77e160366f",
  "createdDate": "2025-04-01T10:15:30Z",
  "lastModifiedDate": "2025-04-10T12:00:00Z",
  "activityTypes": [
    "music",
    "food",
    "family"
  ],
  "registrations": 120,
  "isOwnedByUser": true
}
```


GET `/events` Get events created or registered by current user

Parameters Try it out

No parameters

Responses

Code	Description	Links
200	List of events Media type */* Controls `Accept` header. **Example Value** | Schema `{ "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e", "title": "Summer Festival", "description": "A day full of music, food, and fun.", "dateTime": "2025-07-15T14:00:00", "duration": "PT5H", "location": "Central Park", "capacity": 500, "imageKey": "events/123456/image.png", "organizerId": "7fa459ea-ee8a-3ca4-894e-db77e160366f", "createdDate": "2025-04-01T10:15:30Z", "lastModifiedDate": "2025-04-10T12:00:00Z", "activityTypes": ["music", "food", "family"], "registrations": 120, "isOwnedByUser": true }`	*No links*

POST /events Create a new event

Creates an event with metadata and an image. Requires multipart/form-data.

Parameters

Try it out

No parameters

Request body — multipart/form-data

event * required — Event metadata

`object`

image * required
string($binary)

Image file for the event

Responses

Code	Description	Links
201	Event created successfully Media type */* Controls `Accept` header. **Example Value** | Schema `{ "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e", "title": "Summer Festival", "description": "A day full of music, food, and fun.", "dateTime": "2025-07-15T14:00:00", "duration": "PT5H", "location": "Central Park", "capacity": 500, "imageKey": "events/123456/image.png", "organizerId": "7fa459ea-ee8a-3ca4-894e-db77e160366f", "createdDate": "2025-04-01T10:15:30Z", "lastModifiedDate": "2025-04-10T12:00:00Z", "activityTypes": ["music", "food", "family"], "registrations": 120, "isOwnedByUser": true }`	*No links*
400	Validation error Media type */* **Example Value** | Schema `{ "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e", "title": "Summer Festival", "description": "A day full of music, food, and fun.", "dateTime": "2025-07-15T14:00:00", "duration": "PT5H", "location": "Central Park", "capacity": 500, "imageKey": "events/123456/image.png", "organizerId": "7fa459ea-ee8a-3ca4-894e-db77e160366f", "createdDate": "2025-04-01T10:15:30Z", "lastModifiedDate": "2025-04-10T12:00:00Z", "activityTypes": ["music", "food",`	*No links*

Code	Description	Links


```
    "family"
  ],
  "registrations": 120,
  "isOwnedByUser": true
}
```


GET `/events/{eventId}` Get event by ID

Returns detailed information about an event.

Parameters Try it out

Name	Description
eventId * required `string($uuid)` *(path)*	UUID of the event to retrieve eventId

Responses

Code	Description	Links
200	Successful retrieval of event Media type */* Controls `Accept` header. **Example Value** | Schema	*No links*


```
{
  "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e",
  "title": "Summer Festival",
  "description": "A day full of music, food, and fun.",
  "dateTime": "2025-07-15T14:00:00",
  "duration": "PT5H",
  "location": "Central Park",
  "capacity": 500,
  "imageKey": "events/123456/image.png",
  "organizerId": "7fa459ea-ee8a-3ca4-894e-db77e160366f",
  "createdDate": "2025-04-01T10:15:30Z",
  "lastModifiedDate": "2025-04-10T12:00:00Z",
  "activityTypes": [
    "music",
    "food",
    "family"
  ],
  "registrations": 120,
  "isOwnedByUser": true,
```

Code	Description	Links


```
  "isRegistered": true,
  "organizerUsername": "organizer_jane"
}
```


404 Event not found — *No links*

Media type

/

Example Value | Schema


```
{
  "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e",
  "title": "Summer Festival",
  "description": "A day full of music, food, and fun.",
  "dateTime": "2025-07-15T14:00:00",
  "duration": "PT5H",
  "location": "Central Park",
  "capacity": 500,
  "imageKey": "events/123456/image.png",
  "organizerId": "7fa459ea-ee8a-3ca4-894e-db77e160366f",
  "createdDate": "2025-04-01T10:15:30Z",
  "lastModifiedDate": "2025-04-10T12:00:00Z",
  "activityTypes": [
    "music",
    "food",
    "family"
  ],
  "registrations": 120,
  "isOwnedByUser": true,
  "isRegistered": true,
  "organizerUsername": "organizer_jane"
}
```


DELETE `/events/{eventId}` Delete an event by ID

Parameters | Try it out

Name	Description
eventId * required `string($uuid)` *(path)*	UUID of the event to delete eventId

Responses

Code	Description	Links
200	Event deleted successfully	*No links*

Media type

/

Code	Description	Links

Controls `Accept` header.

Example Value | Schema


```
{
  "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e",
  "title": "Summer Festival",
  "description": "A day full of music, food, and fun.",
  "dateTime": "2025-07-15T14:00:00",
  "duration": "PT5H",
  "location": "Central Park",
  "capacity": 500,
  "imageKey": "events/123456/image.png",
  "organizerId": "7fa459ea-ee8a-3ca4-894e-db77e160366f",
  "createdDate": "2025-04-01T10:15:30Z",
  "lastModifiedDate": "2025-04-10T12:00:00Z",
  "activityTypes": [
    "music",
    "food",
    "family"
  ],
  "registrations": 120,
  "isOwnedByUser": true
}
```


404 Event not found — *No links*

Media type

/

Example Value | Schema


```
{
  "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e",
  "title": "Summer Festival",
  "description": "A day full of music, food, and fun.",
  "dateTime": "2025-07-15T14:00:00",
  "duration": "PT5H",
  "location": "Central Park",
  "capacity": 500,
  "imageKey": "events/123456/image.png",
  "organizerId": "7fa459ea-ee8a-3ca4-894e-db77e160366f",
  "createdDate": "2025-04-01T10:15:30Z",
  "lastModifiedDate": "2025-04-10T12:00:00Z",
  "activityTypes": [
    "music",
    "food",
    "family"
  ],
  "registrations": 120,
  "isOwnedByUser": true
}
```


Event Search

Search for events using filters and pagination

GET `/events/search` Search events with filters

Searches for events using optional filters such as location, date range, duration, and category.

Parameters

Try it out

Name	Description

eventFilterRequest * required
`object`
(query)
	Filtering parameters for events
	Example Value | Schema
	{
	"location": "Madrid",
	"durationFrom": "PT0H",
	"durationTo": "PT24H",
	"dateTimeFrom": "2025-06-01T00:00:00",
	"dateTimeTo": "2025-08-01T00:00:00",
	"activityTypes": [
	"sports",
	"music"
	]
	}

page `integer($int32)` *(query)*	Page number (0-based) *Default value* : 0 *Example* : 0 0
size `integer($int32)` *(query)*	Page size *Default value* : 10 *Example* : 10 10

Responses

Code	Description	Links
200	Filtered list of events Media type application/json Controls `Accept` header. **Example Value** | Schema { "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e",	*No links*

Code	Description	Links


```
  "title": "Summer Festival",
  "description": "A day full of music, food, and fun.",
  "dateTime": "2025-07-15T14:00:00",
  "duration": "PT5H",
  "location": "Central Park",
  "capacity": 500,
  "imageKey": "events/123456/image.png",
  "organizerId": "7fa459ea-ee8a-3ca4-894e-db77e160366f",
  "createdDate": "2025-04-01T10:15:30Z",
  "lastModifiedDate": "2025-04-10T12:00:00Z",
  "activityTypes": [
    "music",
    "food",
    "family"
  ],
  "registrations": 120,
  "isOwnedByUser": true
}
```


400	Invalid filter values or pagination parameters	*No links*

Schemas

EventRequest Collapse all `object`

Event metadata

`title`* Collapse all `string` `≥ 1 characters`

Title of the event

Example="Sunset Yoga in the Park"

`description`* Collapse all `string` `≥ 1 characters`

Detailed description of the event

Example="Join us for a relaxing yoga session during sunset."

`dateTime`* Collapse all `string` `date-time`

Start date and time of the event

Example="2025-07-01T18:00:00"

`duration`* Collapse all `string`

Duration of the event in ISO-8601 format

Example="PT2H"

`location`* Collapse all `string` `≥ 1 characters`

Location where the event will take place

Example="Retiro Park, Madrid"

`capacity` Collapse all `integer` `≥ 1` `int32`

Maximum number of participants allowed

Example=30

`activityTypes` Collapse all `array<string>`

List of activity types associated with the event

`Items` `string`

Example Collapse all `array`

#0="yoga"

#1="wellness"

EventResponse Collapse all object

Detailed response model for an event

id Collapse all string uuid

Unique identifier of the event

Example="6fa459ea-ee8a-3ca4-894e-db77e160355e"

title Collapse all string

Title of the event

Example="Summer Festival"

description Collapse all string

Detailed description of the event

Example="A day full of music, food, and fun."

dateTime Collapse all string date-time

Date and time when the event starts

Example="2025-07-15T14:00:00"

duration Collapse all string

Duration of the event in ISO-8601 format

Example="PT5H"

location Collapse all string

Location where the event takes place

Example="Central Park"

capacity Collapse all integer int32

Maximum number of participants allowed

Example=500

imageKey Collapse all string

Storage key or URL for the event image

Example="events/123456/image.png"

organizerId Collapse all string uuid

UUID of the user who organizes the event

Example="7fa459ea-ee8a-3ca4-894e-db77e160366f"

createdDate Collapse all string date-time

Timestamp when the event was created

Example="2025-04-01T10:15:30Z"

lastModifiedDate Collapse all string date-time

Timestamp when the event was last modified

Example="2025-04-10T12:00:00Z"

activityTypes Collapse all array<string>

List of activity types associated with the event

Items string

Example Collapse all array

#0="music"

#1="food"

#2="family"

registrations Collapse all `integer` `int32`

Number of users registered for the event

Example=120

isOwnedByUser Collapse all `boolean`

Whether the event is owned by the current user

Example=true

DetailedEventResponse Collapse all `object`

Detailed event response including registration status and organizer info

id Collapse all `string` `uuid`

Unique identifier of the event

Example="6fa459ea-ee8a-3ca4-894e-db77e160355e"

title Collapse all `string`

Title of the event

Example="Summer Festival"

description Collapse all `string`

Detailed description of the event

Example="A day full of music, food, and fun."

dateTime Collapse all `string` `date-time`

Date and time when the event starts

Example="2025-07-15T14:00:00"

duration Collapse all `string`

Duration of the event in ISO-8601 format

Example="PT5H"

location Collapse all `string`

Location where the event takes place

Example="Central Park"

capacity Collapse all `integer` `int32`

Maximum number of participants allowed

Example=500

imageKey Collapse all `string`

Storage key or URL for the event image

Example="events/123456/image.png"

organizerId Collapse all `string` `uuid`

UUID of the user who organizes the event

Example="7fa459ea-ee8a-3ca4-894e-db77e160366f"

createdDate Collapse all `string` `date-time`

Timestamp when the event was created

Example="2025-04-01T10:15:30Z"

lastModifiedDate Collapse all `string` `date-time`

Timestamp when the event was last modified

Example="2025-04-10T12:00:00Z"

activityTypes Collapse all `array<string>`

List of activity types associated with the event

Items `string`

Example Collapse all `array`

#0="music"

#1="food"

#2="family"

`registrations` Collapse all `integer` `int32`

Number of users registered for the event

Example=120

`isOwnedByUser` Collapse all `boolean`

Whether the event is owned by the current user

Example=true

`isRegistered` Collapse all `boolean`

Indicates if the current user is registered for the event

Example=true

`organizerUsername` Collapse all `string`

Username of the event organizer

Example="organizer_jane"

EventFilterRequest Collapse all `object`

Filters used for searching events

`location` Collapse all `string`

Filter by location or city

Example="Madrid"

`durationFrom` Collapse all `string`

Minimum duration of the event

Example="PT0H"

`durationTo` Collapse all `string`

Maximum duration of the event

Example="PT24H"

`dateTimeFrom` Collapse all `string` `date-time`

Start of the date-time range for the event

Example="2025-06-01T00:00:00"

`dateTimeTo` Collapse all `string` `date-time`

End of the date-time range for the event

Example="2025-08-01T00:00:00"

`activityTypes` Collapse all `array<string>`

List of activity types to filter by

`Items` `string`

Example Collapse all `array`

#0="sports"

#1="music"

B.3 Reviews Service API

Swagger
Supported by SMARTBEAR

/v3/api-docs

Explore

Reviews Service API

OAS 3.1

/v3/api-docs

Servers

http://localhost:8080 - Generated server url

Authorize

Reviews

Operations related to event reviews

POST /reviews Create a new review

Creates a review for an event organizer with rating and comments.

Parameters

Try it out

No parameters

Request body required

application/json

Example Value | Schema

```
{
  "userId": "6fa459ea-ee8a-3ca4-894e-db77e160355e",
  "rating": 4,
  "text": "Very well organized event!"
}
```

Responses

Code	Description	Links
201	Review created successfully	*No links*

Code	Description	Links

Media type

/

Controls `Accept` header.

Example Value | Schema


```
{
  "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e",
  "raterUsername": "jane_doe",
  "username": "john_smith",
  "rating": 4,
  "text": "Great event with excellent organization.",
  "createdAt": "2025-06-01T12:34:56Z",
  "isOwnedByUser": true
}
```


400	Invalid input data	*No links*

GET `/reviews/{id}` Get review by ID

Retrieves a single review by its UUID.

Parameters | Try it out

Name	Description
id * required `string($uuid)` *(path)*	UUID of the review to retrieve id

Responses

Code	Description	Links
200	Review retrieved successfully	*No links*

Media type

/

Controls `Accept` header.

Example Value | Schema


```
{
  "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e",
  "raterUsername": "jane_doe",
```

Code	Description	Links


```
  "username": "john_smith",
  "rating": 4,
  "text": "Great event with excellent organization.",
  "createdAt": "2025-06-01T12:34:56Z",
  "isOwnedByUser": true
}
```


404	Review not found	*No links*

DELETE `/reviews/{id}` Delete a review by ID

Deletes a review identified by its UUID.

Parameters Try it out

Name	Description
id * required `string($uuid)` *(path)*	UUID of the review to delete id

Responses

Code	Description	Links
204	Review deleted successfully	*No links*
404	Review not found	*No links*

GET `/reviews/user/{userId}` Get all reviews for a specific user

Returns all reviews written for the given user ID.

Parameters Try it out

Name	Description
userId * required `string($uuid)`	UUID of the user whose reviews to retrieve

Name	Description
(path)	userId

Responses

Code	Description	Links
200	List of reviews retrieved successfully Media type */* Controls `Accept` header. **Example Value** | Schema `{ "id": "6fa459ea-ee8a-3ca4-894e-db77e160355e", "raterUsername": "jane_doe", "username": "john_smith", "rating": 4, "text": "Great event with excellent organization.", "createdAt": "2025-06-01T12:34:56Z", "isOwnedByUser": true }`	*No links*
404	User not found or no reviews available	*No links*

Schemas

ReviewRequest

Collapse all `object`

Review creation request

userId* Collapse all `string` `uuid`

UUID of the user being reviewed

Example="6fa459ea-ee8a-3ca4-894e-db77e160355e"

rating* Collapse all `integer` `[1, 5]` `int32`

Rating score from 1 to 5

Example=4

text* Collapse all `string` `[0, 1000] characters`

Review comment text

Example="Very well organized event!"

ReviewResponse

Collapse all `object`

Response model for a review

id Collapse all `string` `uuid`

Unique identifier of the review

***Example**="6fa459ea-ee8a-3ca4-894e-db77e160355e"*

raterUsername Collapse all `string`

Username of the user who wrote the review

***Example**="jane_doe"*

username Collapse all `string`

Username of the user who is being reviewed

***Example**="john_smith"*

rating Collapse all `integer` `int32`

Rating score from 1 to 5

***Example**=4*

text Collapse all `string`

Text content of the review

***Example**="Great event with excellent organization."*

createdAt Collapse all `string` `date-time`

Timestamp when the review was created

***Example**="2025-06-01T12:34:56Z"*

isOwnedByUser Collapse all `boolean`

Indicates if the current authenticated user is the owner of this review

***Example**=true*

C Appendix III: JaCoCo Reports

users Sessions

users

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.whatstheplan.users.controller		77%		75%	4	17	7	42	3	15	0	3
com.whatstheplan.users.services		73%		0%	3	12	10	35	0	9	0	2
com.whatstheplan.users.exceptions		50%		n/a	2	4	4	8	2	4	2	4
com.whatstheplan.users.utils		84%		n/a	1	5	0	7	1	5	0	1
com.whatstheplan.users		37%		n/a	1	2	2	3	1	2	0	1
com.whatstheplan.users.model		100%		n/a	0	3	0	46	0	3	0	1
com.whatstheplan.users.config		100%		n/a	0	15	0	29	0	15	0	5
com.whatstheplan.users.model.request		100%		n/a	0	2	0	14	0	2	0	1
com.whatstheplan.users.model.response		100%		n/a	0	2	0	16	0	2	0	2
Total	108 of 1,006	89%	7 of 10	30%	11	62	23	200	7	57	2	20

Created with JaCoCo 0.8.12.202403310830

Figure C.1: JaCoCo report for the Users Service

events Sessions

events

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.whatstheplan.events.controller		92%		100%	2	37	4	75	2	33	0	4
com.whatstheplan.events.services		97%		100%	4	97	2	236	4	94	0	5
com.whatstheplan.events.config		92%		50%	2	22	8	67	1	21	0	7
com.whatstheplan.events.utils		44%		n/a	2	3	4	9	2	3	0	1
com.whatstheplan.events.model.entities		76%		25%	6	11	0	14	0	5	0	3
com.whatstheplan.events.config.s3		76%		n/a	1	3	0	6	1	3	0	1
com.whatstheplan.events		37%		n/a	1	2	2	3	1	2	0	1
com.whatstheplan.events.repository		99%		64%	5	14	0	62	0	7	0	2
com.whatstheplan.events.model		100%		n/a	0	4	0	46	0	4	0	1
com.whatstheplan.events.model.response		100%		n/a	0	2	0	34	0	2	0	2
com.whatstheplan.events.model.request		100%		n/a	0	4	0	28	0	4	0	1
com.whatstheplan.events.client.user		100%		n/a	0	5	0	15	0	5	0	1
com.whatstheplan.events.exceptions		100%		n/a	0	8	0	16	0	8	0	8
com.whatstheplan.events.client.user.config		100%		n/a	0	3	0	12	0	3	0	1
com.whatstheplan.events.model.email		100%		n/a	0	1	0	9	0	1	0	1
Total	116 of 2,882	95%	15 of 42	64%	23	216	20	632	11	195	0	39

Created with JaCoCo 0.8.12.202403310830

Figure C.2: JaCoCo report for the Events Service

reviews Sessions

reviews

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.whatstheplan.reviews.controller		84%		n/a	2	13	4	35	2	13	0	2
com.whatstheplan.reviews.utils		64%		n/a	1	2	0	4	1	2	0	1
com.whatstheplan.reviews.exceptions		61%		n/a	1	3	2	6	1	3	1	3
com.whatstheplan.reviews		37%		n/a	1	2	2	3	1	2	0	1
com.whatstheplan.reviews.config		100%		n/a	0	17	0	38	0	17	0	5
com.whatstheplan.reviews.service		100%		100%	0	9	0	23	0	8	0	1
com.whatstheplan.reviews.client.user		100%		n/a	0	3	0	8	0	3	0	1
com.whatstheplan.reviews.client.user.config		100%		n/a	0	3	0	12	0	3	0	1
com.whatstheplan.reviews.model.response		100%		n/a	0	1	0	9	0	1	0	1
com.whatstheplan.reviews.model.request		100%		n/a	0	1	0	7	0	1	0	1
com.whatstheplan.reviews.repository		100%		n/a	0	1	0	1	0	1	0	1
Total	43 of 665	93%	0 of 2	100%	5	55	8	146	5	54	1	18

Created with JaCoCo 0.8.12.202403310830

Figure C.3: JaCoCo report for the Reviews Service

notifications Sessions

notifications

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.whatstheplan.notifications.service		7%		n/a	1	2	11	12	1	2	0	1
com.whatstheplan.notifications		37%		n/a	1	2	2	3	1	2	0	1
com.whatstheplan.notifications.handler		100%		n/a	0	6	0	29	0	6	0	3
com.whatstheplan.notifications.config		100%		n/a	0	14	0	20	0	14	0	2
com.whatstheplan.notifications.controller		100%		n/a	0	3	0	4	0	3	0	1
Total	54 of 338	84%	0 of 0	n/a	2	27	13	68	2	27	0	8

Created with JaCoCo 0.8.12.202403310830

Figure C.4: JaCoCo report for the Notifications Service