

Booking System for the KaaS Platform

Design Report

Group 14

Authors:

Andrei Niculi

Daniel Actor

Fèlix Navarro Martí

Fernanda Santiago Martinez

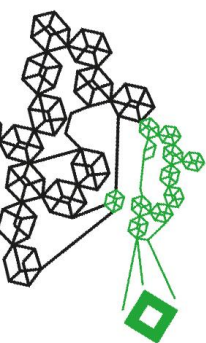
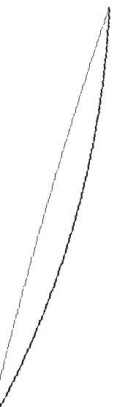
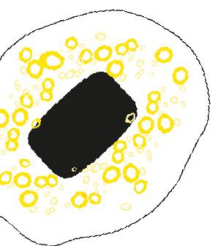
Stefan Pantan

Tudor Ocraïn

Supervisor: Dorus Abeln

Abstract.....	2
1. Introduction	2
2. Domain Analysis.....	3
2.1 Current Situation: Manual booking via spreadsheets	3
2.3 Software Environment and related systems	3
2.4 Issues: Errors, inefficiency, lack of scalability.	4
3. System Proposal.....	4
3.1 Requirements Proposal	4
3.2 Mock-ups / UI sketches	4
3.3 Proposal feedback	5
4. Requirement Specification	5
4.1 Stakeholder Requirements	5
4.2 System Requirements	6
4.3 Quality Requirements (scalability, security, usability)	6
4.4 Requirement Prioritization (MoSCoW).....	6
5. Requirement Analysis	6
5.1 Functional Requirements Analysis.....	7
5.1.1 Must Have Requirements	7
5.1.2 Should Have Requirements	7
5.1.3 Could Have Requirements	7
5.1.4 Could Have Requirement.....	7
5.1.5 Won't Have Requirement (Initial Version)	7
5.2 Non-Functional Requirements Analysis.....	8
5.3 Constraints	8
5.4 Risk Analysis	8
6. Global and Architectural Design	9
6.1 Technology Choices (Backend, Frontend, Database)	9
6.2 Authentication & Access Control	9
6.3 Integration with Provisioning System.....	10
6.4 Architecture	10

6.5 API-first approach	10
7. Detailed Design	11
7.1 Data Model (ER/class diagrams)	11
7.2 Role-based access model	12
7.3 User Interface (wireframes, screenshots)	13
7.4 Scheduling & conflict checking logic	13
7.5 Automated communication (emails, notifications)	14
8. Testing the System	14
8.1 Back-End Testing Plan (unit, integration, usability)	15
8.1.1 Test Scenarios	15
8.1.2 Test Results	15
8.1.3 Risk Analysis and Mitigation	16
8.2 Front-End Testing Plan	17
8.2.1 Test Scenarios	17
8.2.2 Test Results	17
8.2.3 Risk Analysis and Mitigation	18
9. Manuals	18
9.1 User manual	18
9.2 Admin manual	20
9.2.1 Setup	20
9.2.2 Environment variables	20
10. Future Work & Improvements	22
11. Evaluation & Conclusion	22
References	23
Appendices	24
A. Requirement Analysis Tables	24
B. Mock-ups	26
Admin Views	26
Teacher views	27
Student view	29



C. Diagrams30

D. UI Screenshots33

E. Test Scenarios36

Abstract

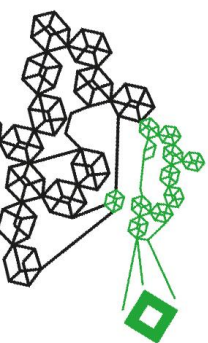
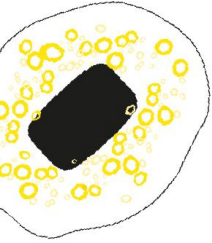
This report is part of the “Design Project” course at the University of Twente. The report presents the design and implementation of an automated booking system for the Kria-as-a-Service (KaaS) platform. The system replaces the current manual spreadsheet-based booking process with a scalable and user-friendly solution. It integrates authentication, user roles, automated scheduling, and provisioning system integration. The report documents the requirements, design choices, implementation, testing, and evaluation of the system. The emphasis of the design is on usability in order to facilitate a better resource booking experience for university staff and students.

1. Introduction

This section introduces the KaaS platform, the problem of manual scheduling, and the goals of the project. It also provides an overview of the report structure.

Kria as a Service (KaaS) is a platform within the University of Twente that grants students, teachers, and administrators access to a pool of 32 Xilinx Kria KV260 FPGA development boards that can be used for educational and experimental purposes. An FPGA development board is a pre-built hardware platform for designing and testing digital circuits, featuring a reprogrammable Field-Programmable Gate Array (FPGA) chip surrounded by essential components like memory and input/output (I/O) ports. These boards allow developers to prototype, test, and deploy custom hardware logic for applications such as signal processing, machine learning, and embedded systems without needing to create a custom printed circuit board (PCB) for every iteration. These boards represent a precious shared resource that deserves an efficient way of reservation that would enhance the user's experience.

The current booking process is done manually, using an online spreadsheet. This process is prone to human error and requires an administrator to regularly check for conflicts, approve or decline requests, and distribute login credentials. Due to the increasing number of people that need to use the boards, the task becomes time-consuming and unscalable. The aim is to offer a fully designed and implemented automated booking system that simplifies the process of gaining access to the FPGA boards. The system combines automated availability checks, role-based authentication, and integration with the provisioning system in order to configure the boards according to the booking details. The



goal is to reduce administrative workload and ensure the user gets a more friendly and reliable experience.

In chapter 2, the current situation and the problem domain will be analyzed in depth. In chapter 3, requirements will be outlined in a formal proposal. Chapters 4 and 5 will delve in depth with the specification and analysis of the requirements. Chapters 6 and 7 elaborate on the architectural and detailed design of the system. Chapter 8 describes testing procedures and expected/actual results. Chapter 9 provides the user manual and the manual for sys-admins. Chapter 10 presents future improvements, and chapter 11 concludes the report with the team's reflection on the process.

2. Domain Analysis

2.1 Current Situation: Manual booking via spreadsheets

Currently, the students and teachers need to make the booking via a shared spreadsheet. They request access to the boards by adding their name and time slots. After this, an administrator has to verify the availability and manually approve bookings. Later, the administrator assigns credentials and notifies the users by email. Additionally, after the completion of a booking, the administrator triggers the reconfiguration process of the boards manually. These whole processes are error-prone, time-consuming, and not scalable.

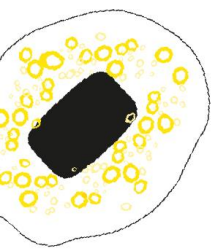
2.2 Stakeholders

The main parties interested in the project are:

- Students: They book the boards for projects or practicals. They need a reliable interface to check the availability of the boards.
- Teachers: They book the boards for lab sessions; their bookings should be automatically approved.
- Administrator: They inspect all bookings, approve requests, and manage hardware provisioning. New tools for monitoring and managing requests are required, also automated conflict resolution.
- System Admins/Support: They maintain the system and provide support for integration with related systems.

2.3 Software Environment and related systems

No specific requirements regarding what technologies to use were made by the client during the elicitation. Therefore, common technologies were chosen, first Django is used as the web framework, Python as a programming language, and the target production server is RedHat Linux.



The new system operates with the university's IT infrastructure, using Single Sign On for secure login and, in the future, a provisioning and recovery system for the automated wiping and configuration of the FPGA boards at the beginning of the booking.

2.4 Issues: Errors, inefficiency, lack of scalability.

The current system is susceptible to human error by default. Administrators constantly need to check for conflicts manually, which can lead to scheduling mistakes. Thus, the method is time-consuming and inefficient because of the tasks that administrators need to conduct verifying availability, accepting/rejecting requests, distributing credentials, notifying users, and triggering the manual reconfiguration of the boards. Moreover, the design is flawed by the lack of scalability it displays. If more boards or users are added to the equation, the administrator's workload becomes unbearable and unreliable.

3. System Proposal

3.1 Requirements Proposal

KaaS (Kria-as-a-Service) provides remote access to FPGA boards (Xilinx Kria KV260) for students and teachers. Currently, bookings are handled via a shared spreadsheet, which is inefficient, error-prone, and lacks automation. The proposed booking system will automate scheduling, integrate with provisioning, and provide role-based access to ensure smooth and conflict-free usage of FPGA resources.

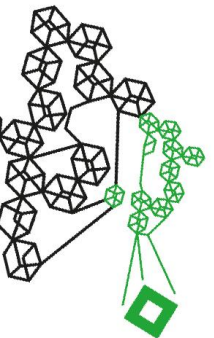
The vision is to create a secure, reliable, and user-friendly booking system that minimizes administrative overhead and ensures students, teachers, and administrators can focus on academic and research activities without friction.

The KaaS Booking System will replace the error-prone manual spreadsheet with a robust, secure, and automated system. Using university SSO, role-based permissions, and integration with the provisioning system ensures smooth academic use of FPGA resources while minimizing admin burden.

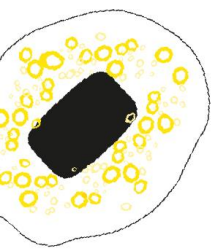
The MoSCoW prioritization ensures a clear roadmap:

- Must-Haves – deliver the core scheduling, authentication, and provisioning.
- Should-Haves – improve classroom use and recover workflows.
- Could-Haves – open the path for future extensibility (APIs, analytics).
- Won't-Haves – keep the scope realistic for the first release

3.2 Mock-ups / UI sketches




The initial interface design follows a calendar-based layout emphasizing clarity and ease of use. Users can view available boards per day, select continuous booking intervals, and visualize availability in real time. Admins have extended capabilities, including blocking



boards for maintenance and managing approvals directly from the interface. The design prioritizes simplicity and consistency with the university's digital environment. All first UI mockups included in the proposal can be found on Appendix B.

3.3 Proposal feedback

Feedback from stakeholders played a crucial role in refining the system requirements and scope. Discussions clarified several expectations and led to significant adjustments:

- 
- **Scope refinement:** advanced functionalities such as provisioning integration, automated board recovery, hardware monitoring, and usage analytics and reporting were decided to be not essential for the initial version of the platform.
 - **UI expectations:** Stakeholder preferred a very simple calendar-style UI that intuitively displays board availability. The platform should allow users to select date intervals, see the number of available boards, and easily identify the soonest possible booking for a specified interval and board count.
 - **Admin controls:** it was emphasized that admins should be able to view and manage all bookings, as well as blocking boards for maintenance and assign permissions to users based on their role.
 - **Usability and Resource allocation:** specific board assignment order is not critical, but continuous boards usage is preferred.
 - **Development focus:** Stakeholder expresses that delivering a functional and reliable core platform is more important than implementing every specified feature. A modular architecture, especially an API design with separate frontend and backend, was encouraged to support maintenance, and future improvements/integrations.

Overall, the feedback reinforced the need to balance functionality, usability, and technical feasibility. The final proposal now reflects a realistic scope aligned with the module timeline and the stakeholder priorities while having a solid foundation for future development.



4. Requirement Specification

4.1 Stakeholder Requirements

The KaaS system targets the following groups at the University of Twente: The supervisor has the highest role of authority and acts as the system administrator. Teachers can automatically approve their bookings, reserve boards for students, and add notes to existing reservations. Students can only request bookings and check availability, but the requests must be approved by someone with higher privileges. All users are informed via email about the booking status and deadlines.

However, the manager can edit the permissions and assign new roles to specific users.

4.2 System Requirements

KaaS needs to provide a simple and reliable way to book Kria boards for any integrant of the University of Twente through the website. The system must show Krias availability, prevent double bookings, and send automatic email confirmations. It should support flexible role management so that administrators can assign permissions as needed. The computers are integrated with the university login system for security purposes and are automatically reimaged when a booking ends or has password resets prior to a new booking. The web application is deployed within the university network and stores data in a secure database.

4.3 Quality Requirements (scalability, security, usability)

The system is designed to manage up to 32 Krias with the possibility of expanding later if needed. Security is maintained by the university's single-in login system and encryption of all sensitive data. The website is clean and intuitive, making it easy to find and book available boards. It should confirm bookings in a relatively short period of time and ensure that no double allocations don't occur.

4.4 Requirement Prioritization (MoSCoW)

The following list represents the distribution of main requirements among the MoSCoW categories:

- Must Have: University SSO authentication, booking creation and confirmation, role management, conflict-free allocation, and cloud deployment.
- Should Have: Classroom bookings and dashboard views depending on user role.
- Could Have: A REST API for future extensions or integrations.
- Won't Have: Sub-day scheduling (the minimum booking unit is one day).

5. Requirement Analysis

The KaaS Booking System aims to automate the allocation and management of FPGA boards for academic use at the University of Twente. The current manual spreadsheet-based approach is inefficient, error-prone, and difficult to maintain. The proposed system introduces a secure, automated, and role-based booking platform that reduces administrative overload.

This section provides a detailed analysis of the project's functional and non-functional requirements, identifies key technical and organizational constraints, and outlines associated risks that may affect the development and deployment phases.



5.1 Functional Requirements Analysis

Functional requirements define the behavior of the system, what the platform must do to satisfy user and stakeholder needs. Requirements have been prioritized using the MoSCoW framework.

5.1.1 Must Have Requirements

Refer to Table 1 on Appendix A for the Must Have Functional Requirements.

5.1.2 Should Have Requirements

These enhance usability and scalability but are secondary to system correctness.

- Classroom bookings for multiple students under a teacher account
- Dashboard views adapted to each user's role

5.1.3 Could Have Requirements

These enable future extensions/integrations and possible institutional reporting once the core system's functionality is stable.

- RESTful API for future integrations
- Monitoring and analytics for usage statistics and error detection

5.1.4 Could Have Requirement

A RESTful API to enable future extensions/integrations

5.1.5 Won't Have Requirement (Initial Version)

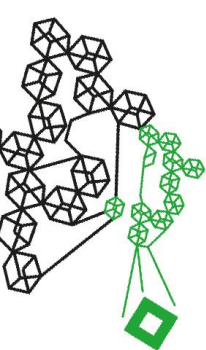
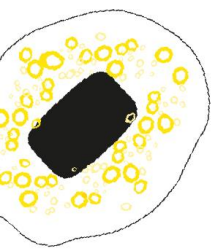
Deferring these features keeps the project feasible within the module timeframe and available hardware constraints.

- Analytics for usage statistics
- Monitoring for system health checks and automated fault detection
- Hardware power management (PDU control)
- Sub-day scheduling
- Integration with provisioning system
- Board recovery triggered by admin

5.2 Non-Functional Requirements Analysis

Non-functional requirements define the system's quality attributes.

The specification of the functional requirements for this project along with their description and implementation strategy can be found on Table 2 on Appendix A.




5.3 Constraints

Type	Constraint	Impact
Organizational	Must run on UT's Red Hat server and authenticate via Active Directory.	Limit infrastructure choices. Containerization is optional but would be beneficial for portability.
Technical	New technical skills for team members. Integration with provisioning API is not available on the first version.	The design must separate the booking logic from provisioning triggers to avoid blocking functionality but still think ahead for a future integration.
Time	Development occurs within a 10-week period with a team of 6 members.	Requires parallel development (frontend and backend) and incremental datelines.
Security/Privacy	Must comply with university IT policies.	User data and access logs must be handled carefully.
Operational	There are limited numbers of boards	Conflict resolution and scheduling logic must ensure fairness and priority enforcement
Dependency	Reliance on SSO	Learn how to implement university SSO on the platform

5.4 Risk Analysis

Risk	Description	Likelihood	Impact
SSO Integration Failure	Misconfiguration of OIDC or lack of implementation instructions leading to login issues.	Medium	High
Role Misconfiguration	Incorrect permissions expose sensitive data or allow unauthorized actions	Medium	High
Booking Conflict Logic Errors	Overlapping bookings under load	Medium	High
Data Corruption or Loss	Database inconsistencies or accidental deletion	Low	High
Performance delays	Slow queries when multiple users book simultaneously	Medium	Medium



Email Delivery Failure	Notification errors might delay or fail to deliver confirmation mails	Low	Medium
Provisioning API Instability	Future integration with the provisioning system might introduce failures or delays.	Medium	Medium
Schedule or Coordination Risks	Dividing the team into two sub teams working on the frontend and backend on two different branches might face uneven progress or merge conflicts	Medium	Medium

6. Global and Architectural Design

6.1 Technology Choices (Backend, Frontend, Database)

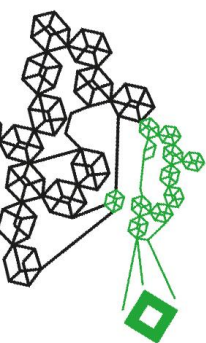
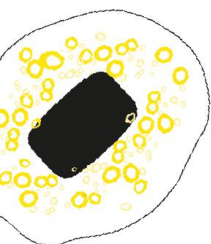
The project is built using Vue/Vite for the frontend, Django and DjangoRestFramework for the backend/api and Nginx as a reverse proxy. For the database, PostgreSQL was chosen because of its robustness and being very commonly used. If ever another database system is necessary, then it is easy to switch, because all the SQL queries are performed by the Django ORM.

6.2 Authentication & Access Control

Users can get access to the booking interface by logging in with their UTwente account via single sign-on. The tokens provided by Microsoft are then used by the web server to request or send data to the API. This is all done via the standard Microsoft OIDC flow according to their [documentation](#). The API is also separately accessible by third-party applications via the API keys. Non-admin users can only see their bookings, while admins have the ability to view all the bookings currently in the system. Admins also have the ability to view and manage the boards and internal attributes of users. Not their email and password, but their roles and the permissions each role has. It is also possible to create new roles, but for custom permissions, changes need to be made in the code for them to be fully functional.

6.3 Integration with Provisioning System

The integration with the provisioning system was not a priority at the top of our list because it was not in development at the time we developed this system. Because of this, we thought the best approach would be to provide a solid base for future integration between the two systems. We did this by providing access to the API using API-keys, meaning that any future applications that need to get information from or interact with the backend can do so by accessing the specific endpoints with the API-key.



6.4 Architecture

The overall system architecture follows a modular, service-oriented design that separates concerns between the frontend, backend, and data layers. This modularity supports maintainability, scalability, and possible future integration with other university systems, such as the provisioning system.

At a high level, the architecture consists of the following components:

1. **Database (PostgreSQL):** The relational database stores data for users, bookings, and boards. It's accessed through Django's ORM.
2. **Backend / API Layer (Django and Django REST framework):**
The Django backend exposes REST API points that handle authentication, booking logic and validation, role-based permissions checks, and email notifications. Each request passes through authentication of middleware that validates API keys to access protected resources/views. The backend operates independently of the frontend, to ensure future systems/tools can reuse the same API.
3. **Reverse proxy (Nginx):** Nginx is the gateway between external request and internal services. It routes API requests to Django backend and forwards frontend routes to Vue application.
4. **Frontend (Vue):** Provides an interactive calendar-based interface for users to view board availability, create bookings, and receive feedback in real time. It communicates with the backend through REST API endpoints.
5. **Authentication Service (Microsoft OIDC):** User authentication is via UT Single Sign-ON using the Microsoft OpenID Connect flow.

6.5 API-first approach

The system is developed following an API first methodology, which means the backend API functions as a central communication layer for the web interface and any future third-party application. This design choice was made due to the many benefits it provides:

- **Separation of concerns:** the frontend and backend are separated, allowing independent development and testing. Backend development can continue in parallel while frontend development can mock API responses.
- **Reusability and expandability:** any future integrations such as a mobile application, the provisioning system, or board monitoring can directly access the same REST endpoints via secure API keys without needing to change the architecture.
- **Maintainability:** A clear API design standardizes data exchange formats and improves code clarity, making it easy to maintain.



7. Detailed Design

7.1 Data Model (ER/class diagrams)

The data model follows a relational structure implemented through Django, allowing a clear map between database entities and Python classes. It enforces logical constraints at the database and application level, such as automated cascade deletion when users or boards are removed.

The main entities with their relationships are illustrated in the *KaaS Class Diagram* and the *Database Schema Diagram* (see Figure 7 & 8 on Appendix C).

- **User:** authenticated UT users signed in with university credentials through SSO. Each user has an associated role (Student, Teacher, or Admin) stored in the Role model.
- **Role and Permissions:** Roles define access rights and tasks' permissions within the platform. Roles are stored in a dedicated table and associated with users via a foreign key or group membership.
- **Board:** represents each FPGA board in the lab. Each board has a unique number and status (available, maintenance, down).
- **Booking:** core entity connecting users and boards. Each booking has a start date, end date, board quantity, associated user, and a current state (requested, pending approval, approved, allocated, active, completed, rejected). Constraints ensure $\text{endDate} \geq \text{startDate}$ and $\text{quantity} \geq 1$.

7.2 Role-based access model

The role-based access model dictates how each user's role interacts with the platform and its API. For the corresponding role-based Use-case diagram, see Figure 8 on Appendix C. Access control is implemented via Django Groups and Permissions, extended by a custom Role model linked to CustomUser.

Students

- Can browse board availability, request and manage their own bookings
- Bookings are always created in the pending approval state and require review and approval by an admin
- Can modify their own booking (extend or cancel) if it still pending or approved
- Receive automated email notifications for booking confirmations, 24 hour remainder, credentials, and expiration warning.

Teachers

- Have all of student permissions
- Can book on behalf of students (e.g., for classroom projects)

Admins

- Have full system access
- Can view and manage all booking, users, and roles and permissions
- Can allocate boards manually if required
- Can access all system logs

In the Django backend, access control is implemented through a combination of custom permission classes and role checks within each ViewSet. For example, to verify the user's role before assigning a status to their booking.

All sensitive endpoints, including board management, approvals, and role editing, are protected by the permission classes, ensuring least-privilege access.

7.3 User Interface (wireframes, screenshots)

The User Interface developed in Vue/Vite has a clean and functional design, which prioritizes simplicity, responsiveness, and ease of use, ensuring accessibility to all users while consistently providing visual feedback on booking status.

The main views of the platform are the following:

- **Login Page:** redirects to UTwente's SSO portal using the Microsoft OIDC flow.
- **Dashboard/Calendar View:** displays an interactive calendar showing the number of boards that are available each day. Selecting an interval shows continuous board availability and automatically suggests the earliest possible start date for the user's request.
- **Request List:** list all student bookings pending for approval with options for the admin to approve or reject and add notes.
- **Boards Dashboard:** lists all 32 boards' information.
- **Bookings Dashboard:** list all previous accepted booking requests with a filtrate option


Screenshots of all platform views can be found in Appendix D.

7.4 Scheduling & conflict checking logic

The scheduling and conflict logic are two of the core components of the system, as they guarantee conflict-free allocation of the limited boards.

The algorithm for board allocation works as follows:

- **Availability check:** when a user selects a date range and quantity, the backend computes the total available boards per day by finding all the bookings that overlap with the specified time period, checking how many boards are booked by those bookings on each day and subtracting that number from the total number of boards.
- **Continuity computation:** identifies the earliest continuous window that satisfies the requested duration and quantity, prioritizing contiguous board use.

- 
- **Priority resolution:** the system applies a predefined priority hierarchy: *Admin > Teacher > Student*. Lower-priority bookings may be rejected or kept in a pending state if capacity is insufficient.
 - **Maintenance handling:** blocks can be put on maintenance state, dynamically affecting allocation results.
 - **Approval and Finalization:** Admin bookings are auto-approved; student and teacher bookings are added to the approval queue. Upon approval, boards are assigned, and when booking starts, credentials are sent to the user.
 - **Job scheduling:** Using Django-APScheduler, background jobs trigger actions such as sending booking reminders or interacting with the provisioning API at booking start and end times.

7.5 Automated communication (emails, notifications)

The implemented automated notification framework enhances the overall user experience, reduces communication delays, and minimizes the manual follow-up burden.

Notification Workflow:

- Booking confirmation/rejection: sent immediately after approval/rejection, including allocated board IDs, time interval, and status.
- Booking autoapproved email: automatic confirmation for teachers and system administrators.
- Booking update: in case the sysadmin modifies something in a booking.
- Booking deletion: announcement if a booking is getting deleted.

Implementation:

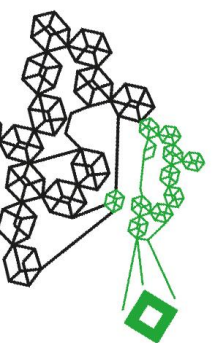
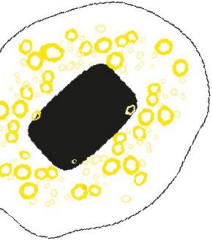
The `reject_booking()` and `perform_create()` methods in `views.py` handle dynamic message generation using templates and helper functions for confirmations.

Scheduled reminders are created via APScheduler jobs, ensuring notifications are reliable even if the main server restarts.

8. Testing the System

This chapter is dedicated to explaining the test plan and analyzing the test results. The functionalities that are tested are highlighted, and the approach is explained. Moreover, risks and mitigations are indicated.

The testing strategy used Python's "PyTest" framework and Django's built-in "TestCase" and "APIClient", to properly simulate database interactions and API requests.



8.1 Back-End Testing Plan (unit, integration, usability)

The multi-layered test plan covers unit, integration, and API level usability testing to guarantee correctness and reliability of the system.

Unit Tests: The focus is on the smallest blocks of logic in the project. In the file “test_models.py” unit tests that cover the correctness of the core data constraints of the models are present. The scope was to ensure that the database correctly applies to the integrity rules of the models, like preventing a booking from ending before it starts.

Integration Tests: Most tests were done at this level. These types of tests ensure that different components of the project work together. For these tests, Django’s “APIClient” was used to simulate the exchange of user and machine request-response cycles. The parts tested were:

- Validating the permission logic against the view logic. Permissions and ownership for different types of users are tested.
- Confirming that API views trigger the correct email notifications
- Ensuring the correct and complete workflow of the provisioning system.

Usability Tests: Due to the API -first nature of the design of the backend, traditional usability testing was swapped for API usability. Thus, the API endpoints were designed to be restful and predictable. Moreover, invalid requests are accompanied by informative messages, such as “Board with id ... not found”, to provide help for developers when integrating the API.

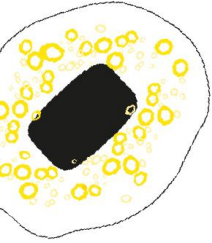
8.1.1 Test Scenarios

The test plan was executed through a series of test scenarios. The scenarios were derived directly from the project’s requirements. To run these tests, the tester needs to open a terminal and be at the project location, then run this command: “pytest backend/api/tests -v”

See Appendix E, Table 3 for the complete Test Scenarios table.

8.1.2 Test Results

The tests were conducted in the order described in the Test Scenario table (Table 3). The Provisioning API file failed initially by returning “404 Not Found” errors instead of 200, 400 and 403 as expected. This indicated a flaw in machine-to-machine integration. After analyzing the code, the investigation revealed two issues. First, the test client was making a request to an invalid URL (“/provision”), instead of the correct one (“/api/provision”). The second fault was a view method mismatch, as the router used to redirect pages, was mapping POST requests to a method named “create”. However, the view in the file “views.py” had this method named “post.”



The issues were corrected by updating the “testProvisionApiKey.py” file to use the afferent URL, and the “post” method in “TriggerProvisioningViewSet” was renamed to “create” to align with the default router’s expectations. Furthermore, with these corrections applied, all tests passed successfully, thus confirming that all functionalities, from data integrity to security and integration, were working as specified.

8.1.3 Risk Analysis and Mitigation

Potential vulnerabilities were identified after conducting a risk analysis. The outcome was ensuring that mitigations were in place and tested.

Risk01:

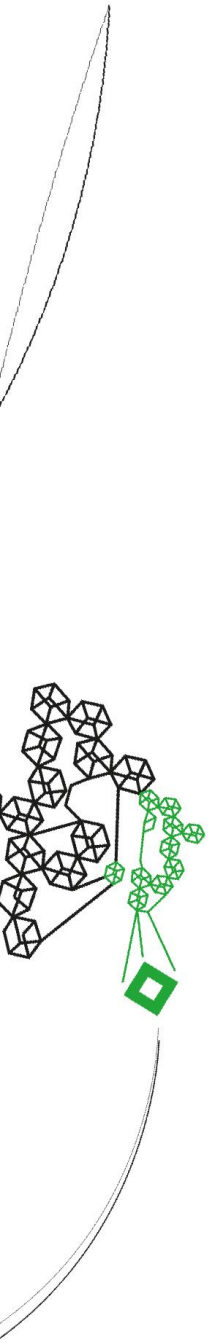
- **Risk Description: Unauthorized Booking Access.** A bad-intentioned user could modify another user's booking by guessing URL parameters.
- **Mitigation Strategy:** The file “permissions.py” (cite: kaas-booking-system/backend/api/permissions.py), was implemented by using “DjangoModelPermissionWithOwnership”, a custom permission class that checks ownership of the booking object.
- **Proof:** the test file “test_object_permission_with_ownership” checks whether the permission correctly blocks other users from viewing a booking owned by a specific user.

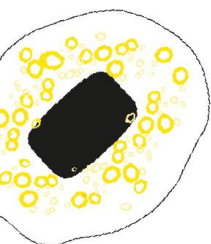
Risk02:

- **Risk Description: Unauthorized Provisioning.** An external service could call the provisioning API endpoint and wipe/change the password to one of the boards.
- **Mitigation Strategy:** Implemented a “HasApiKey” permission key that requires a valid, secret API key for all requests inside the “views.py” in order to secure the “TriggerProvisioningViewSet”.
- **Proof:** The test cases “test_invalid_api_key” and “test_missing_api_key” inside the “testProvision_ApiKey.py” confirmed that all requests without a valid key are rejected with a “403 Forbidden error”.

Risk03

- **Risk Description: Booking Overlaps.** This is a matter of data integrity, where a conflict could arise when two users book the same board at the same time.
- **Mitigation Strategy:** The BookingSerializer inside “serializers.py” has a custom “validate” method that queries the database for conflicting bookings, before the creation of a new one.
- **Proof:** A manual test confirms that sending a second overlapping request is rejected by a “400 Bad Request” error code.





Risk 04

- **Risk Description: Invalid Dates.** The problem stands at the creation of a booking that ends before it starts.
- **Mitigation Strategy:** Data integrity of the booking model inside “models.py” has been enforced by the addition of the “end_not_before_start” CheckConstraint.
- **Proof:** the test “test_start_before_end_constraint” checks whether the creation of such booking is possible

8.2 Front-End Testing Plan

Alongside back-end testing, the client-side booking operations were validated using front-end testing.

The frontend of the KaaS booking system was tested using Vitest and @testing-library/vue to verify both the UI behavior of key views and the correctness of the client-side service layer that talks to the Django API. Tests are written in TypeScript and run in a browser-like environment configured with setup file.

8.2.1 Test Scenarios

These tests ensure that the main booking flows, dashboards, and service calls behave as intended and that the frontend receives and calls the API commands contract defined in the backend.

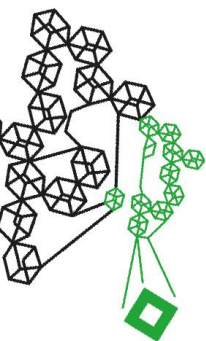
See Appendix E, Table 4 for the complete Frontend Test Scenarios table.

8.2.2 Test Results

The BookingPage component demonstrate robust functionality with 15 passing tests covering core booking system features. The test suite successfully verifies calendar rendering, form validation, and user interaction workflows while maintaining isolation through comprehensive service mocking. Key components including the BookingCalendar, modal dialogs, and form inputs are thoroughly tested for proper rendering and behavior. API integration tests confirm correct data flow between the frontend and backend services, with proper error handling for network failures. The testing strategy ensures reliable user experience across date selection, availability checking, and booking submission processes while maintaining test stability through controlled mock environments.

8.2.3 Risk Analysis and Mitigation

High-risk scenarios include board overallocation beyond the 32-board capacity and role-based access control failures. The test suite mitigates these through capacity validation and permission enforcement in student/teacher booking flows. Critical booking conflict risks



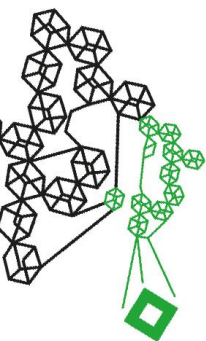
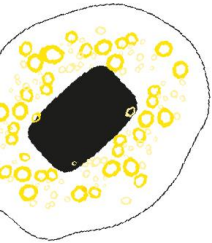
are partially addressed by calendar availability tests, though concurrent booking scenarios require additional coverage.

9. Manuals

9.1 User manual

Log in:

- To access the web application, all users must go directly to the main page and click the log in button. They later need to provide their UT credentials to verify their identity.
- Once the user has successfully log in he will be able to access a range of sections within the web application
 - Booking Page:
 - Access to this page is global
 - Users can book Krias by selecting the period of time they want to reserve them. Once this is done, a little window for extra specifications will show up. In there, users can introduce the following information:
 - Starting booking date
 - Ending booking date
 - Amount of Krias needed
 - Notes to explain the purpose of the booking
 - To prevent cases where users can't find specific periods of time with a determinate number of free Krias, users can make use of the Find Soonest Availability button. When they click on it, a little menu will show up where users can introduce the period of time they want to book X number of boards, and the system will automatically search for them the soonest range of days when the desired booking can be made.
 - Once a booking is made, a confirmation email will be sent to the user. In case the user doesn't have permission to get his bookings instantly accepted, they will need to wait for other users with higher role permissions to accept or reject it (a second email will be sent to the user once the decision has been made)
 - Booking Dashboard:
 - Access to this page is restricted by user role permissions
 - Users can see a list with all current requests made by users pending acceptance with the following information:
 - How many users participate in this booking
 - Number of boards required
 - Period this booking takes place
 - Status of the booking




- At the right part of each element in the list users can interact with 2 different buttons with the following symbols:
 - +: Accept booking
 - x: Reject booking
- Access to this page is restricted by user role permissions
- Non-admins can only see their own bookings, while admins can see all bookings of everyone.
- A list with all previous booking requests is provided with the following information for each element in the list:
 - How many users participate in this booking and their respective roles
 - Number of boards required
 - Period this booking takes place
 - Days left
 - Status of the booking
 - Booking status (Accepted, Rejected, Pending).
- An option to filter results is provided with some element lists on the top right corner of the screen
- An option to clear all inactive bookings is available at the top right corner of the page.
- Boards Dashboard:
 - Access to this page is restricted by user role permissions
 - A list with all the Krias boards supported by the web application is provided with the following information for each element in the list:
 - Its ID number (currently goes from 1 to 32)
 - Status (In use, Maintenance, Not in use)
- User Menu
 - Access to this page is restricted by user role permissions
 - A list of all users that have signed in to the application is provided with the following information for each element in the list:
 - The user's local username. Usually set to their email address but can be manually changed.
 - The user's email address.
 - The current role of the user.
 - A dropdown menu where you can change the role of the user, including admins.

These menus can be seen in figures 10-14 of Appendix B.



9.2 Admin manual

9.2.1 Setup



To set up and run the project using Docker, first ensure that Docker and Docker Compose are installed on your system. Navigate to the root directory of the project. Run the command “docker compose up –build” to build and start all services as defined in the docker-compose.yml file. The backend and frontend will be available at the addresses shown in the terminal output, typically <http://localhost:8000/> for the backend and <http://localhost:5173/> for the frontend, but these may be different depending on how you setup the cloudflare tunnel and/or nginx.

To stop the services, press Control+C in the terminal and then run “docker compose down”. If you make changes to dependencies or code and need to rebuild the containers, use “docker compose up --build” again.


For first-time setup, the database will be initialized automatically if using Docker. No admin accounts are created by default. To create an admin account, run “docker compose exec backend python manage.py createsuperuser” and follow the prompts. For detailed instructions see README.md

Other useful commands include running database migrations with “docker compose exec backend python manage.py migrate” and accessing the backend container shell with “docker compose exec backend bash”.



9.2.2 Environment variables

Variable	Purpose	Example/Default Value
SECRET_KEY	Django cryptographic key	django-insecure-...
DEBUG	Enable debug mode (False in production)	True (dev), False (prod)
ALLOWED_HOSTS	Allowed hostnames for requests	localhost, 127.0.0.1, kaas.daniel.actor
CSRF_TRUSTED_ORIGINS	Trusted origins for CSRF protection	https://kaas.daniel.actor
POSTGRES_DB	Database name	django_db

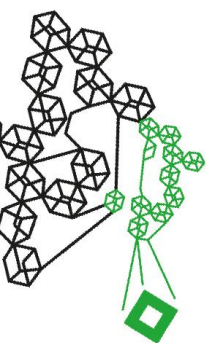
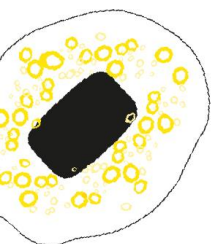


POSTGRES_USER	Database user	django_user
POSTGRES_PASSWORD	Database password	django
POSTGRES_HOST	Database host/service name	db
POSTGRES_PORT	Database port	5432
OIDC_ISSUER	OIDC issuer URL (for SSO)	<a href="https://login.microsoftonline.com/<tenant-id>/v2.0">https://login.microsoftonline.com/<tenant-id>/v2.0
OIDC_AUDIENCE	OIDC client/application ID	<client-id>
OIDC_JWKS_URL	(Optional) JWKS endpoint override	(leave blank for auto-discovery)
EMAIL_BACKEND	Email backend (console for dev, SMTP for prod)	django.core.mail.backends.console.EmailBackend
EMAIL_HOST	SMTP server host (prod only)	smtp.example.com
EMAIL_PORT	SMTP server port (prod only)	587
EMAIL_HOST_USER	SMTP username (prod only)	user@example.com
EMAIL_HOST_PASSWORD	SMTP password (prod only)	password
EMAIL_USE_TLS	Enable TLS for SMTP (prod only)	True

10. Future Work & Improvements

In the next list, some implementations and upgrades to the system will be presented as future work.

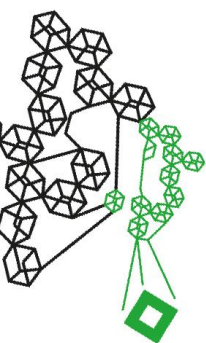
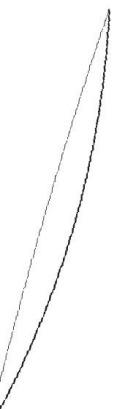
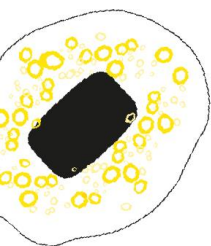
- **Mobile deployment:** The current project has only been tested in the objective device (PC) but an implementation on different devices such as mobile phones will help to increase the product usability
- **Provisioning system integration:**
- **CLI implementation:** This will increase the manager's domain, allowing him to have control over features inaccessible through the current admin interface.



- **Roles menu:** One thing that was not implemented was a menu to change and add roles. As there is a custom model made for roles, it is easy to extend roles to the applications' needs. Adding this menu would allow the admin more possibilities in managing the users.
- **More role attributes:** As for now, the permissions the admin can assign to each role are limited; future updates could increase the number of possible attributes for new roles.
- **API connection to Krias:** The implementation of an API could be used to interact with Krias directly (e.g. power any Kria down).
- **GUI customization:** In the current version, no visual customization is implemented. Implementing different distributions and colors could improve the average user's experience, since they could adjust the interface that suits them more.
- **Shorter booking periods:** For now, bookings are divided by days, which are the minimal amount of time possible. Reducing it to hours or minutes could avoid unnecessary time loss and better organization of bookings.
- **Data analysis:** Detecting and analyzing important information such as what percentage of bookings represent each course would help detecting system limitations and help developing and upgrading the system capabilities efficiently
- **Token storage:** Currently, the tokens sent by Microsoft are stored in Sessionstorage in the browser. This is not the most secure way, as the tokens are accessible by Javascript, so if an xss-attack happened the tokens would be accesible. A more secure way would be to use a so-called "Backend for Frontend" (BFF) pattern. This means that only the backend would see the tokens and authentication is communicated using secure cookies.

11. Evaluation & Conclusion

The design of the KaaS booking system followed an iterative approach based on the stakeholder needs and the feasibility of the project within the time constraint. The project evolved from the initial concept of replacing a manual spreadsheet with a more structured, automated, and scalable platform. The design process consisted of defining a clear problem space, analyzing requirements with the MoSCoW method, and gradually transforming them into a coherent system architecture and detailed implementation plan. Early design stages consisted of understanding user needs and addressing institutional constraints. Through the design process, the importance of adopting an API-first design was determined, which not only simplified frontend and backend development but also laid the groundwork for future integrations. The project was developed collaboratively by a six-member team, separated into two teams, one to focus on the frontend and one on the backend development. Collaboration



relied heavily on GitLab for version control and issue tracking, enabling parallel development across both teams without major merge conflicts. Weekly team and supervisor meetings provided checkpoints to synchronize progress and adjust priorities as the module evolved. Knowledge sharing also played a central role: members alternated in reviewing pull requests, debugging issues, testing features, and writing documentation and reflection, which not only distributed the workload but also increased collective understanding of the entire system.

Project management was guided by agile principles, with weekly milestones. We employed short feedback loops to validate the design decisions early and frequently.

The MoSCoW prioritization framework proved effective in keeping the project scope realistic within the time frame, helping the team distinguish between essential functionality and desirable future features.

Time constraints presented an ongoing challenge. Integrating UT’s SSO authentication, role-based permissions, booking logic, and deployment proved to be specially challenging and required close coordination, and some initially planned functionalities had to be postponed. Nevertheless, the team delivered a final working product, supported by automated testing and manual validation of critical features such as booking conflict resolution and email notifications.

References

Django (<https://www.djangoproject.com>)
Docker (<https://www.docker.com>)
Nginx (<https://nginx.org>)
Postgres (<https://www.postgresql.org>)
Vite (<https://vite.dev>)
uv (<https://docs.astral.sh/uv>)
Microsoft OpenID (<https://learn.microsoft.com/en-us/entra/identity-platform/v2-protocols-oidc>)


Appendices

A. Requirement Analysis Tables

Table 1

Must Have Functional Requirements

Category	Requirement	Analysis
----------	-------------	----------



University SSO Authentication	Integration with university Single Sign-On (SSO)	Critical for compliance with UT's IT security policies and to remove the need for separate user management. It ensures secure login and unified credentials for students, teachers, and admins.
User Roles and Access Control	Students require admin approval for bookings; teachers' bookings are auto approved; admins can approve/reject, override, and manage roles and users.	Role-based permissions are essential for separating allowed tasks. It will be implemented using Django's Groups/ Permissions system extended by a custom Role model.
Booking Interface	Calendar-based interface to select start/end dates, board quantity, and easily visualize boards' availability.	Central interaction point for all end users. Must balance simplicity with accurate real-time availability.
Conflict-Free Allocation	Prevent double bookings and automatically prioritize Admin > Teacher > Student.	Requires backend logic to ensure correct booking transactions and database constraints.
Automated Emails	Automatic emails for confirmation, reminders (24 h before), end-of-booking warnings, and any booking changes.	Ensures transparency and user awareness. Implemented using Django's email backend and scheduler jobs.
Teacher and Admin Operations	Teachers can book for other students. Admins can manage maintenance periods and view and change all bookings.	Enables efficient resource management and allows for any necessary manual intervention.
Cloud deployment	Platform should support deployment on UT-managed cloud servers to enable scalability, maintainability, and continuous accessibility.	This would allow dynamic resource allocation, remote access, and a future easier integration with the provisioning system.

Table 2
Functional Requirements

Category	Requirement	Implementation Strategy
Security (Must)	Role-based authorization, University SSO, hashed boards' password, and secure token handling.	Handled by Django. Critical for compliance with institutional cybersecurity standards.
Reliability (Must)	System ensures no double allocations and the integrity of booking records.	Database constraints and transactions ensure consistency.
Usability (Must)	Clear and simple UI, accessible to non-technical users.	Achieved through Vue.js and validated via peer feedback.
Scalability (Should)	Must support at least 32 boards with growth potential.	Horizontal scaling is possible via database optimization.
Performance (Should)	Booking confirmation and availability checks in less than 2s.	Requires caching or efficient query design.
Maintainability (Should)	Clear code separation: backend (Django) and frontend (Vue).	Facilitates modular development and maintenance.
Accessibility (Could)	WCAG-compliant interface.	Improves inclusivity for visually impaired users. Planned for later phases.

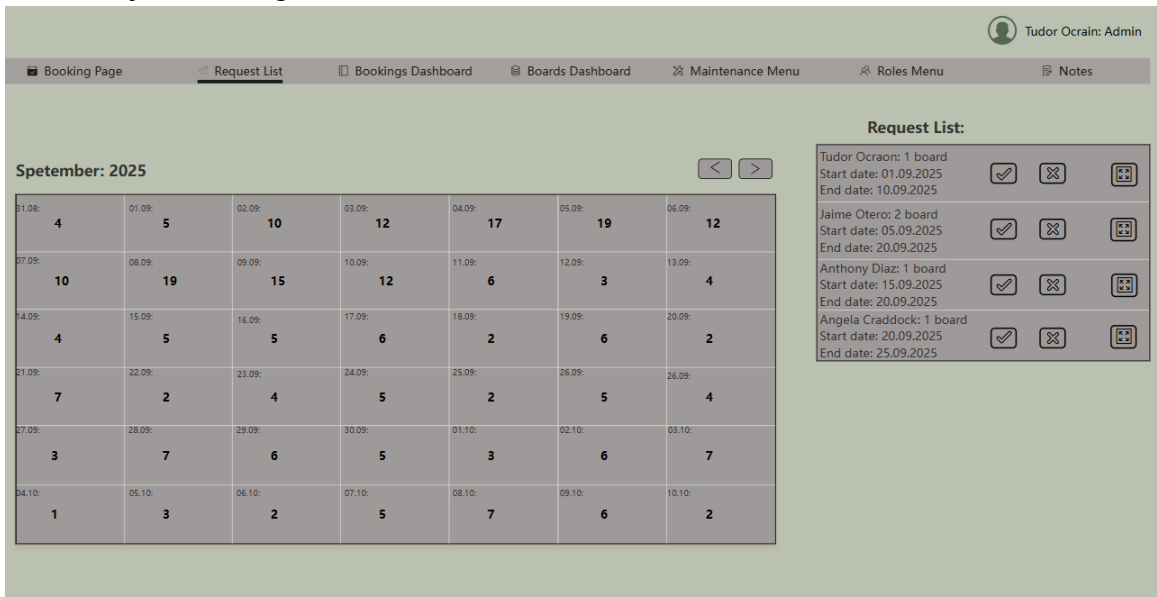
B. Mock-ups

Admin Views

Figure 1
Admin Booking Page



Figure 2
Admin Request List Page



Teacher views
Figure 3
Teacher Booking pop-up window

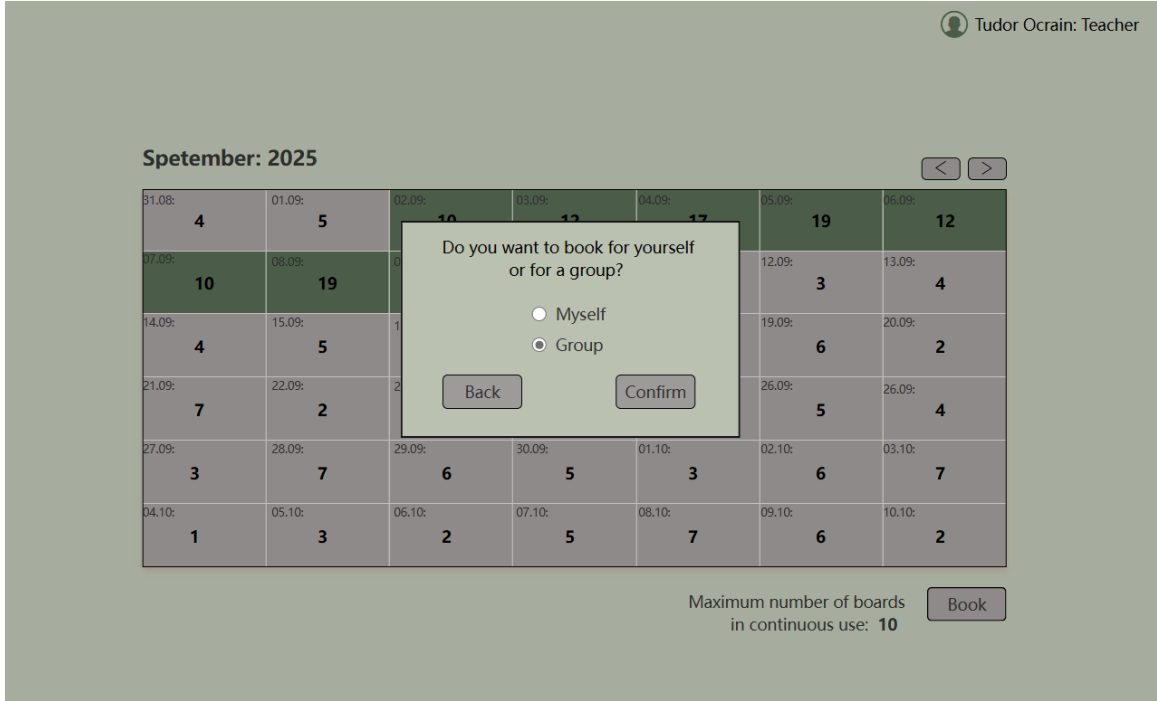


Figure 4
Teacher Group Booking pop-up window

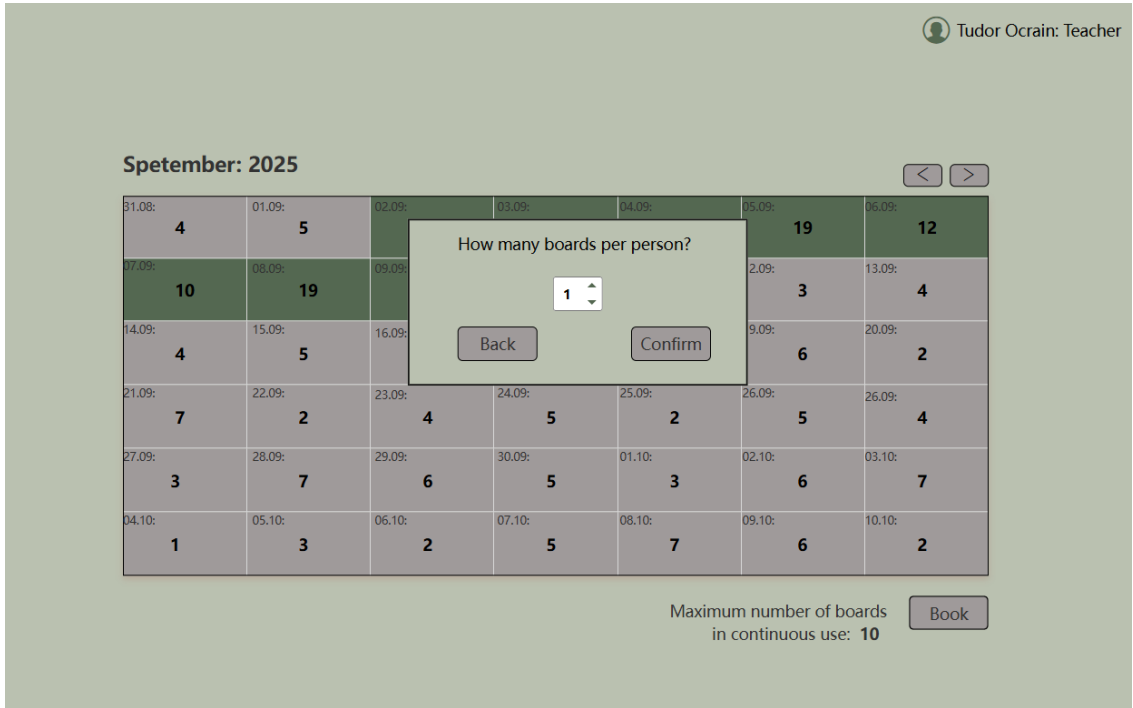
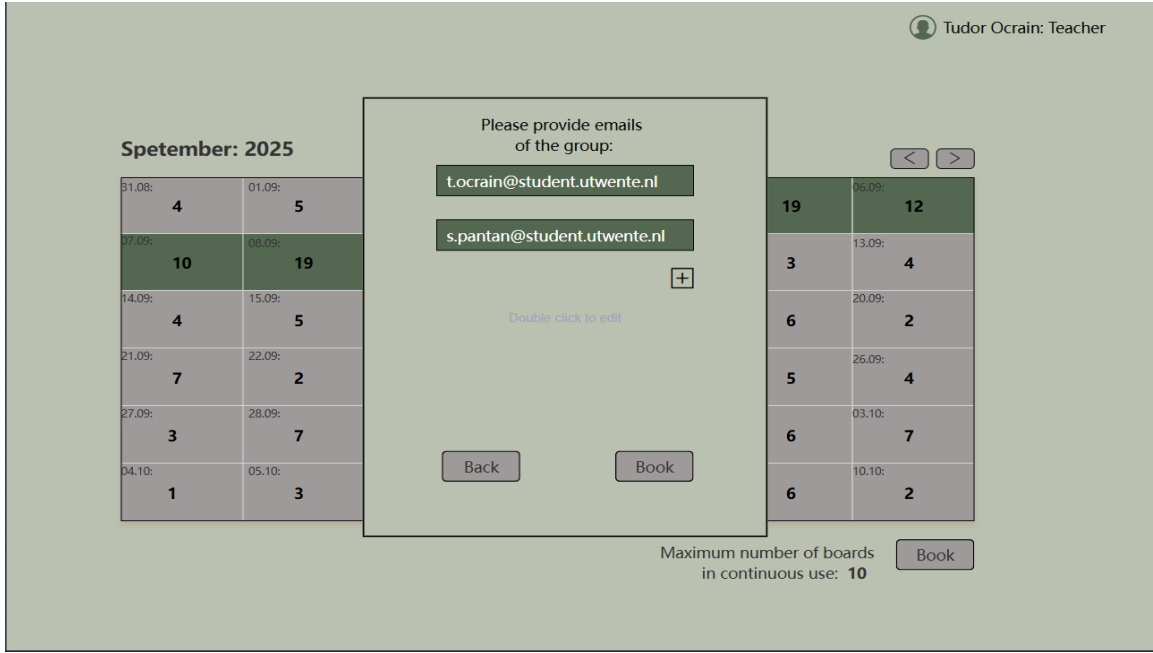
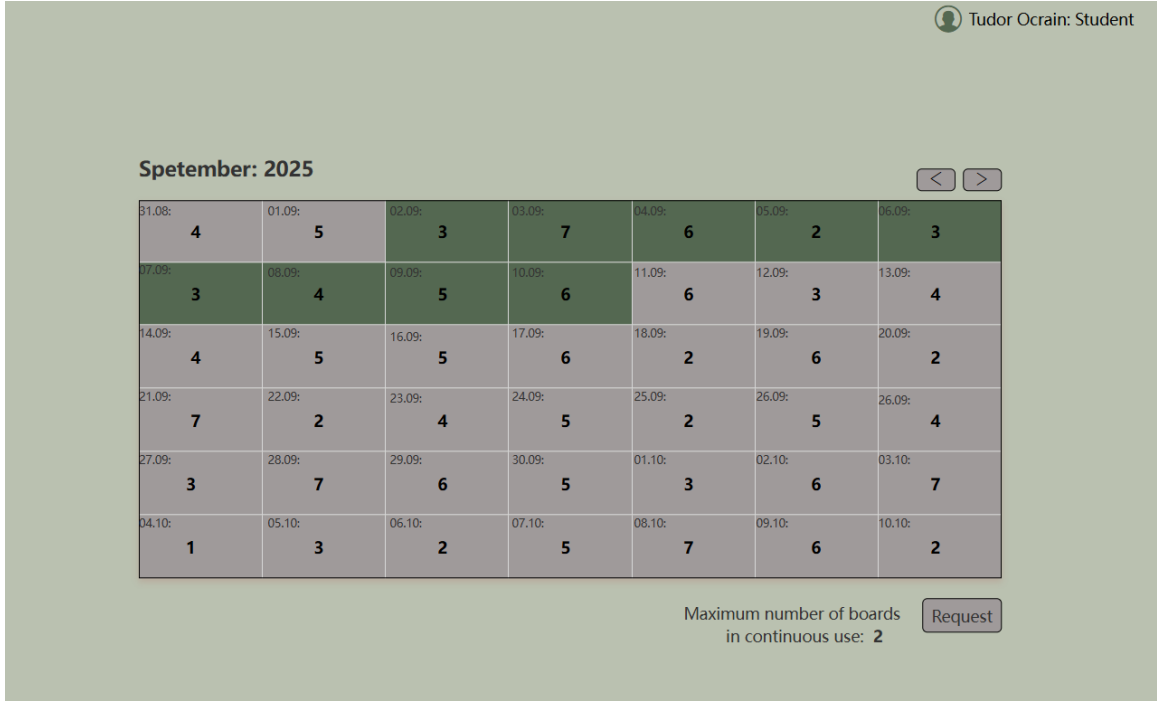
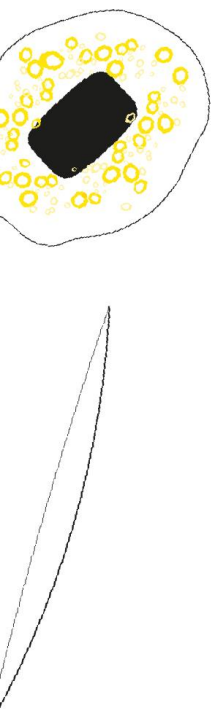


Figure 5
Teacher Group Booking final pop-up window



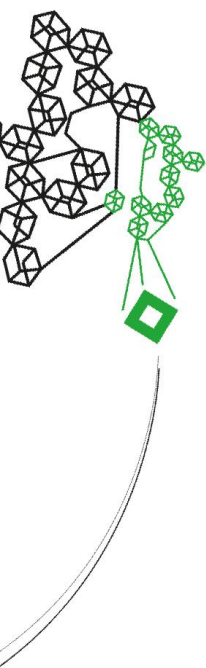
Student view

Figure 6
Student homepage



C. Diagrams

Figure 7
Microsoft OAuth 2.0 Authorization Flow



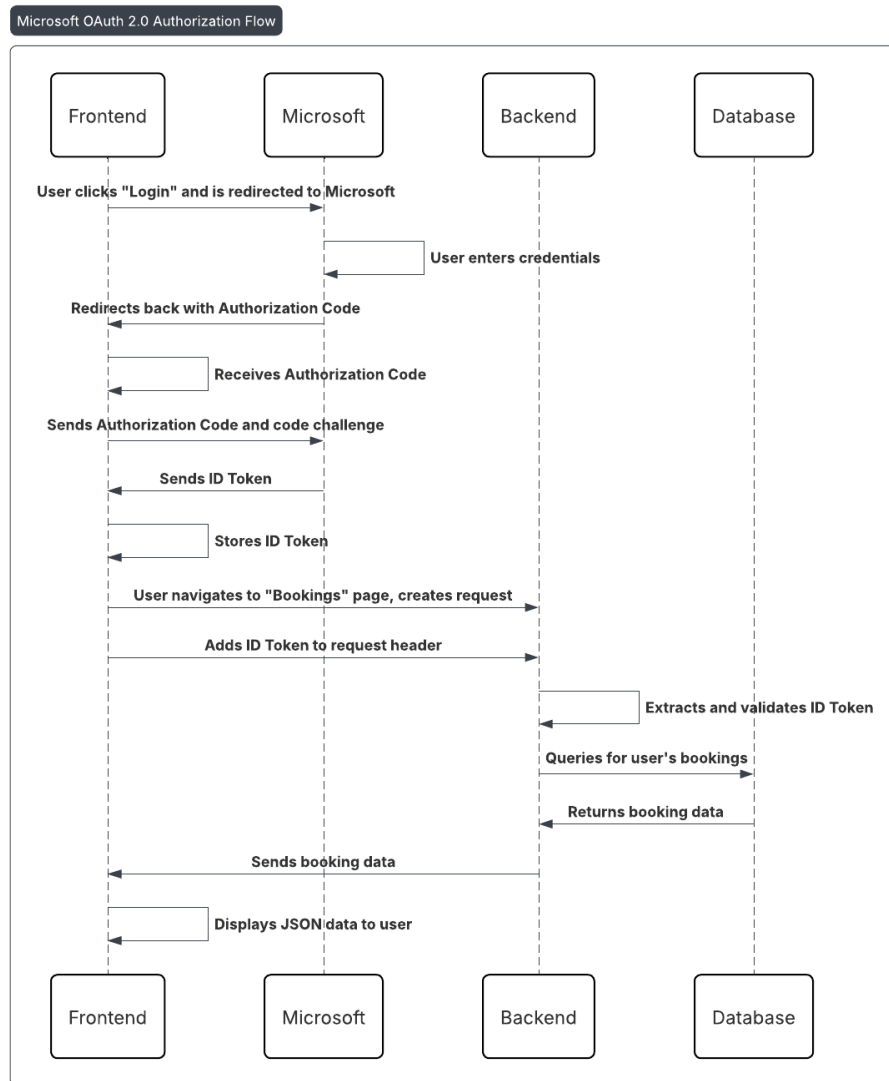
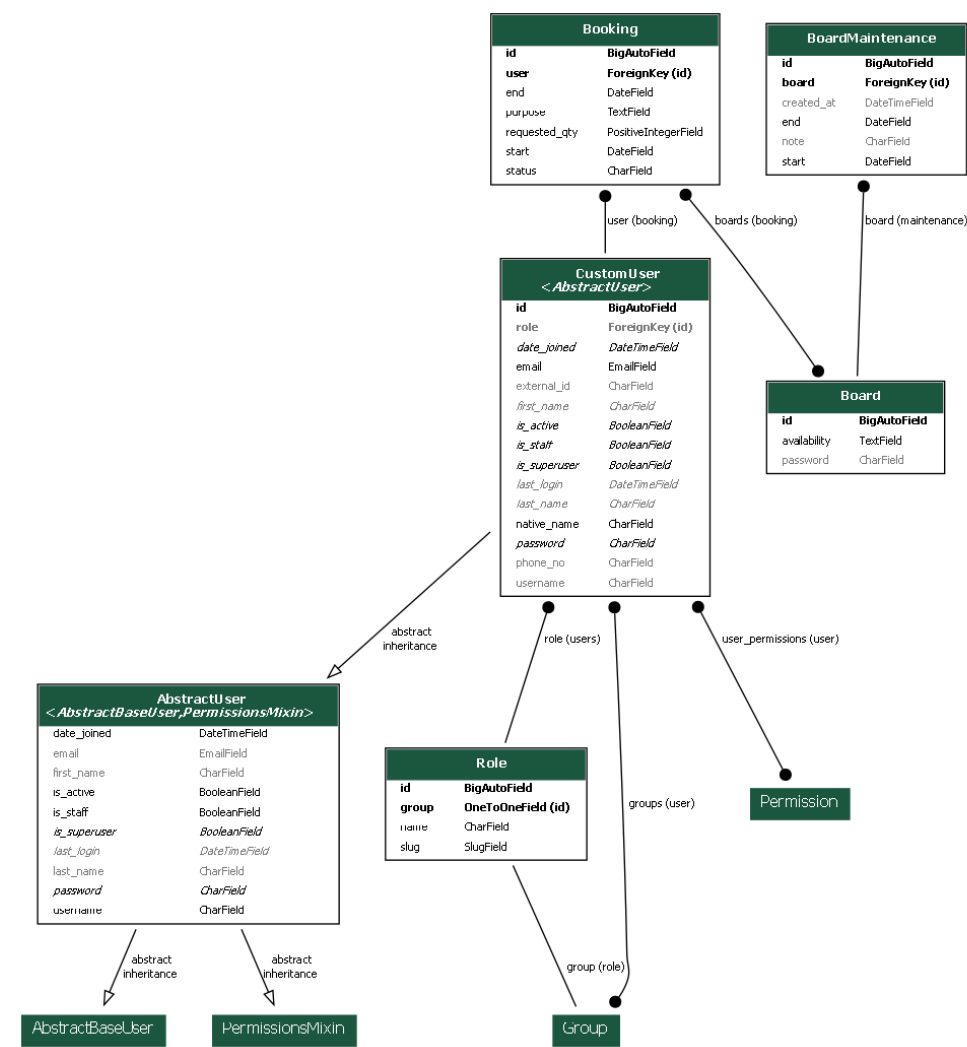


Figure 8
Database Schema diagram



(generated using django-extensions and pydotplus, based on models.py)

Figure 9
Role-based Use-case diagram

Homepage

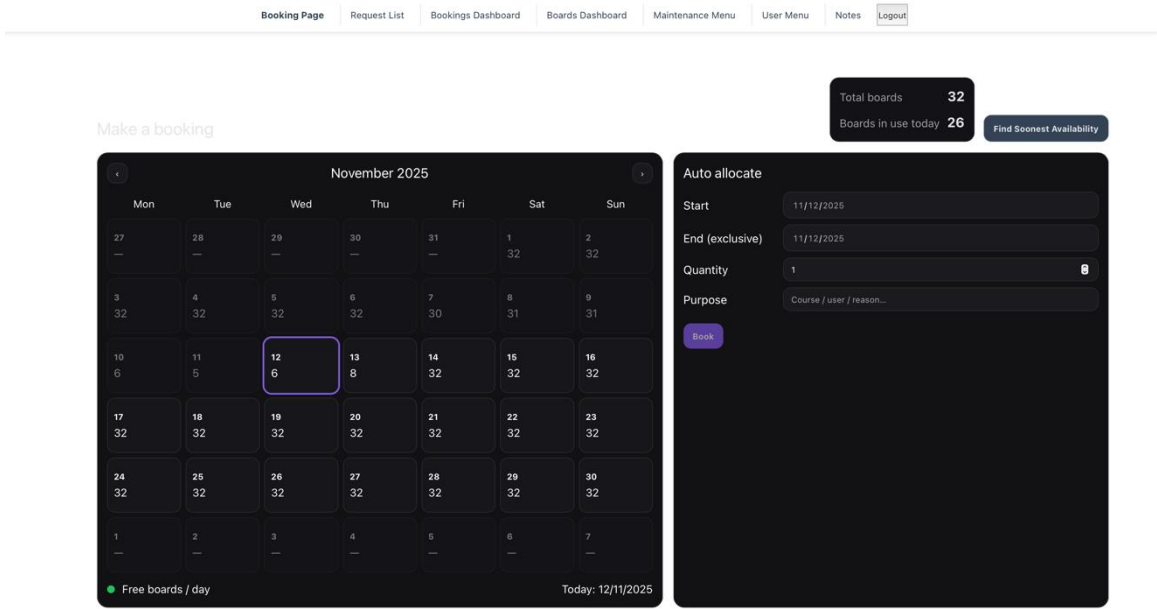


Figure 11

Request list page

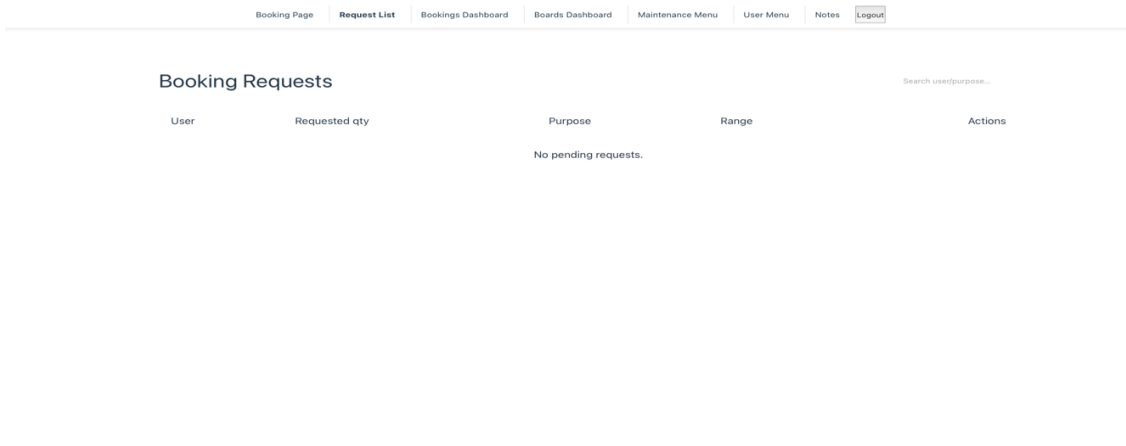


Figure 12
Booking Dashboard

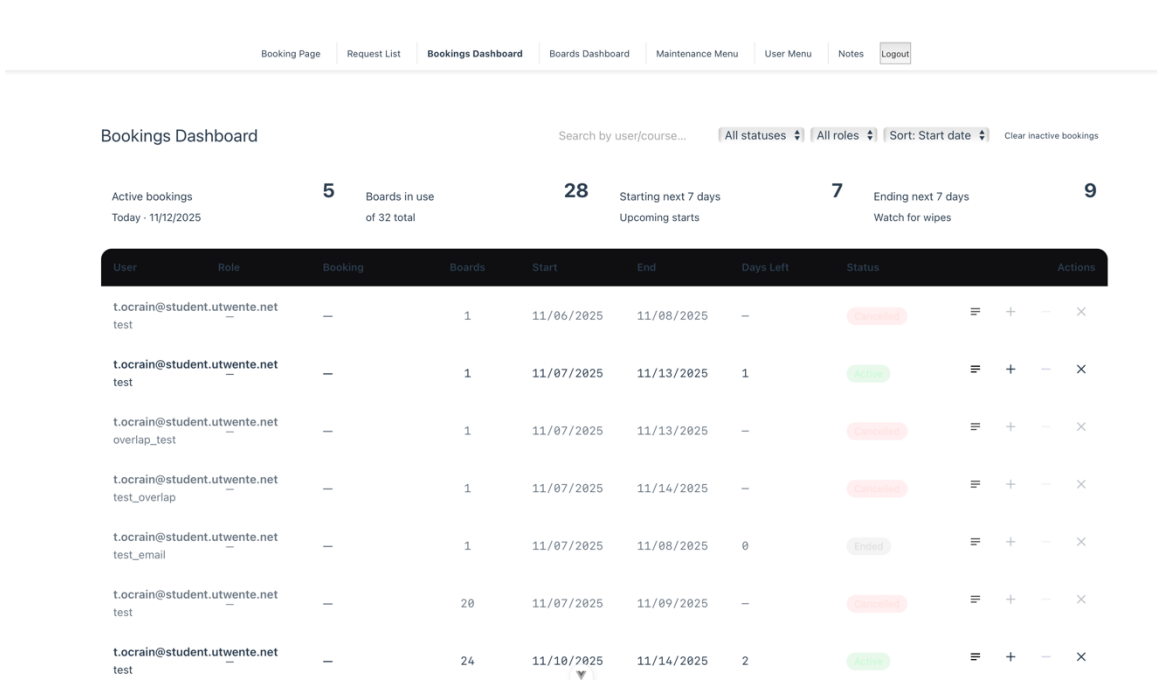


Figure 13
Boards Dashboard page

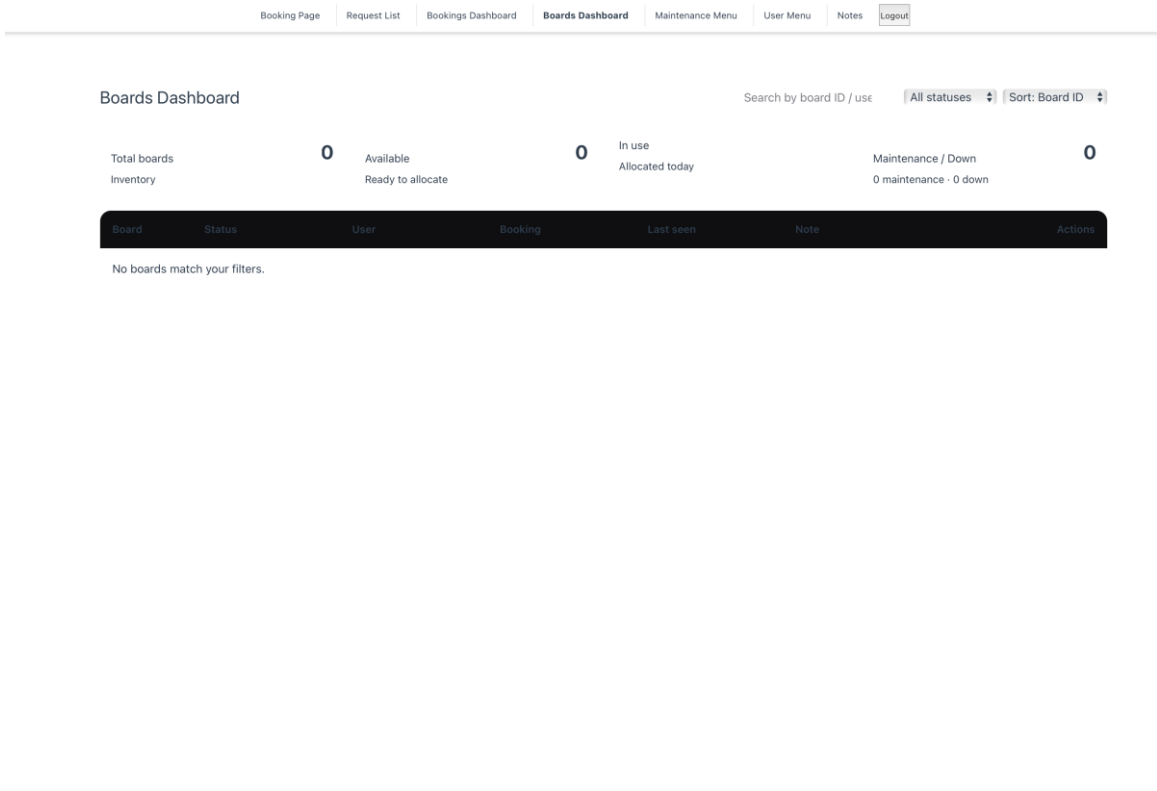
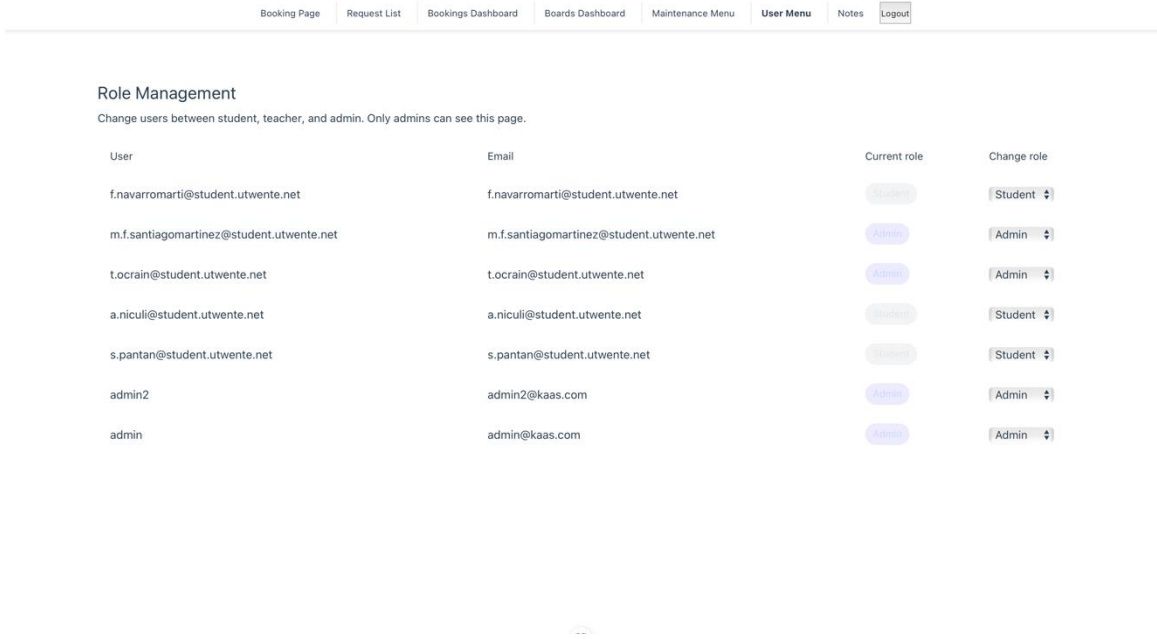



Figure 14
User Menu page



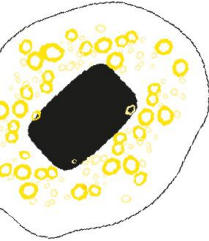
E. Test Scenarios

Table 3
Backend Test Scenarios

Module	Test File	Test Scenario	Requirement tested
Data Models	test_models.py	test_create_booking.py	Verifies that a standard booking can be created
		Test_start_before_end_constraint	Confirms that the database raises an IntegrityError if a booking's end date is before its start date.
Permissions	Test_permissions.py	Test_group_permission	Verifies that the IsStudent, IsTeacher, IsManager, and IsAdmin permission classes work as expected.
		test_object_permission_with_ownership	Confirms that a user (e.g., a Student) can only view their own bookings and not bookings belonging to other users.




Email System	test_email_notifications.py	test_send_student_pending_email	Verifies the correct "pending" email is sent when a Student creates a booking.
		test_send_teacher_auto_approved_email	Verifies the correct "confirmed" email is sent when a Teacher creates a booking.
		test_send_student_approved_email	Verifies the correct email is sent when an Admin approves a booking.
		test_send_student_rejected_email	Verifies the correct email is sent (including the reason) when an Admin rejects a booking.
		test_send_booking_updated_email	Confirms that a notification is sent when a booking's details are changed.
Provisioning API	testProvisionApiKey.py	test_provisioning_success	(Integration Test) Confirms that a POST request with a valid API Key successfully provisions a board and updates its password/status in the database.
		test_invalid_api_key	Confirms that a request with an <i>invalid</i> API Key is rejected with a 403 Forbidden status.
		test_missing_api_key	Confirms that a request with no API Key is rejected with a 403 Forbidden status.
		test_board_not_found	Confirms that a request to provision a non-existent board ID is rejected with a 404 Not Found status.

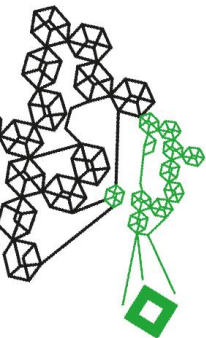



		test_invalid_payload	Confirms that a request missing required data (e.g., new_password) is rejected with a 400 Bad Request status.
--	--	----------------------	---

Table 4
Frontend Test Scenarios



Module	Test File	Test Scenario	Requirement Tested
Boards Dashboard	BoardsDashboard.spec.ts	Render the Boards Dashboard view and verify the “Boards Dashboard” title	Boards dashboard loads successfully and exposes a clear entry point for board monitoring and management
Booking Page	BookingPage.spec.ts	Render the Booking Page and verify the “Make a booking” heading and Book button	Booking UI is visible and the main booking action is clearly available to the user
		Fill Start/End/Quantity/Purpose and click Book, checking autoBook is called with the correct attributes	Frontend correctly constructs and submits a booking request (dates and quantity) to the backend API
Booking Services	bookingServices.spec.ts	Call fetchAvailability and check URL, query parameters, and returned data	Client correctly queries /bookings/availability/ with start/end parameters and returns backend data





		Call autoBook and verify HTTP method, headers, body, and response handling	Booking service a POST with a JSON body that matches the booking payload and returns the response
Test Infrastructure	setup.ts	Stub alert and confirm and load @testing-library/jest-dom	