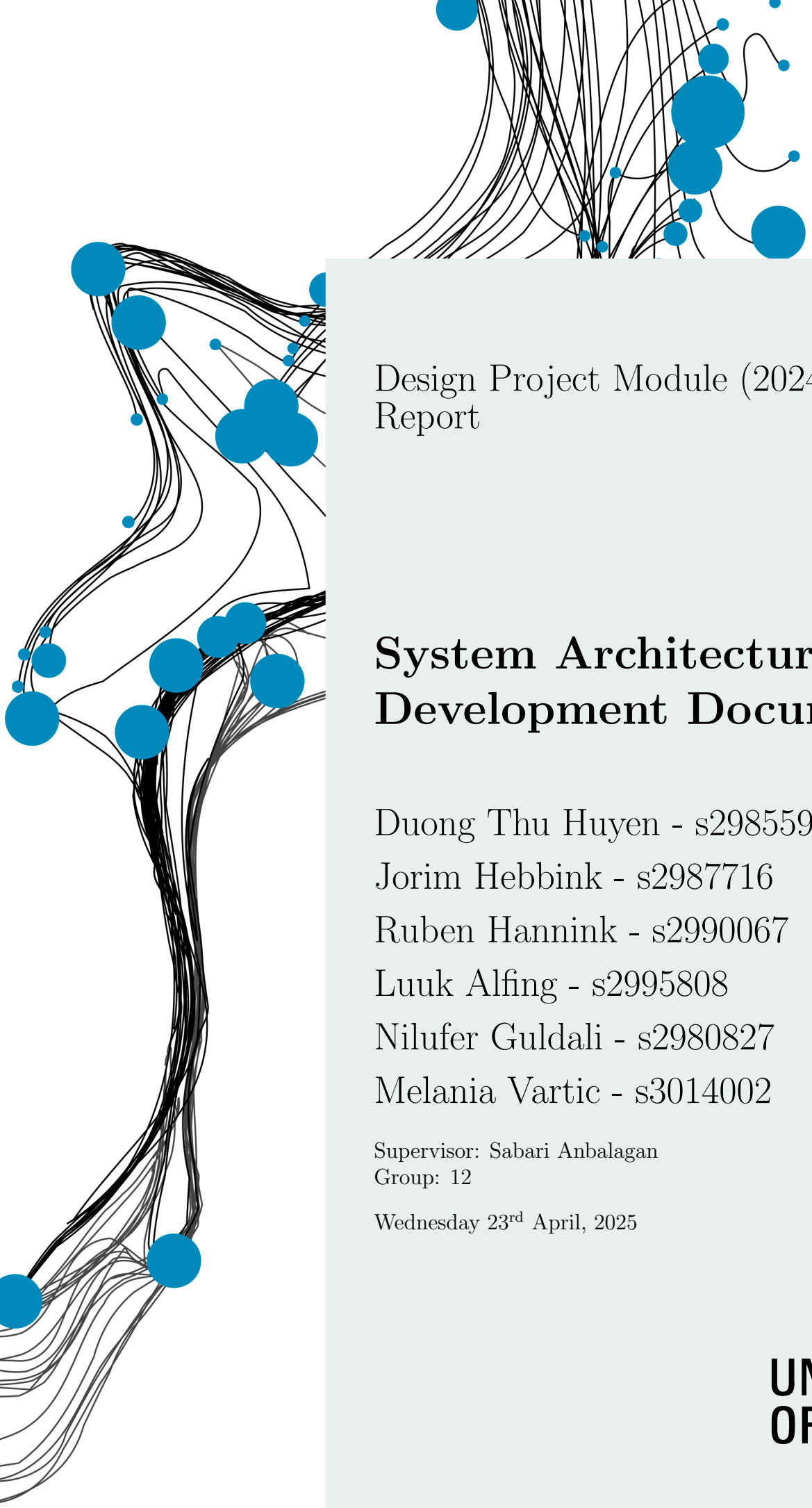


Design Project Module (202400005)
Report

System Architecture and Development Document

Duong Thu Huyen - s2985594

Jorim Hebbink - s2987716

Ruben Hannink - s2990067

Luuk Alfing - s2995808

Nilufer Guldali - s2980827

Melania Vartic - s3014002

Supervisor: Sabari Anbalagan
Group: 12

Wednesday 23rd April, 2025

UNIVERSITY OF TWENTE.

Contents

Abstract

The Pervasive Systems Research Group manages and maintains a variety of electronic components. To monitor the hardware, an Inventory System must be designed. This application aims to assist with hardware allocation management and the borrowing process. This document contains the key findings discovered during the development process of a system to manage the hardware of Pervasive Systems. The report provides an overview of the existing problem to be addressed, details on coordination with the client, functional and quality requirements, user stories, implementation trajectory, literature research, planning, along with insights into the development process, testing, and results. In addition, it will explain in depth the design choices and frameworks used, provide a security analysis, and compare the system with existing solutions.

Glossary

Abbreviation	Meaning
AI	Artificial Intelligence
API	Application Programming Interface
CI/CD	Continuous integration and continuous deployment
CORS	Cross-Origin Resource Sharing
CRUD	Create, read, update and delete
DDOS	Distributed Denial of Service
DOS	Denial of Service
DTO	Data Transfer Object
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IMS	Inventory Management System
JPA	Java Persistence API
JSON	JavaScript Object Notation
JWT	JSON Web Token
OAS3	OpenApi Specification version 3.1
OTP	One Time Password
PS	Pervasive Systems
SQL	Structured Query Language
UML	Unified Modeling Language
URL	Uniform Resource Locator
XSS	Cross-Site Scripting

Table 1: Abbreviations and their meaning

AI Statement

Artificial intelligence has been used for this project and documentation in the context of explaining core concepts and auto-correction. Throughout the development process, when the team found out about some new technology or needed more insight into a concept, in some of the cases, artificial intelligence was used. Moreover, the platform this document was written in contains an embedded AI corrector, namely Writefull, that checks and highlights spelling errors or certain inconsistencies in writing.

Chapter 1

Context

1.1 Pervasive Systems Research Group

Pervasive Systems is a research group within the University of Twente that specializes in networked smart sensing systems. Pervasive systems consist of interconnected sensing, computing, and reasoning components deeply embedded in the environment, enabling accessible and unobtrusive interaction. Their goal is to achieve seamless integration of computing with the physical world through sensors, actuators, and computing systems, contributing to environmental sustainability. The Pervasive Systems department maintains hardware such as sensors, Arduino boards, transducers, and other electronic components. Researchers and doctoral students utilize the hardware primarily for research purposes, and therefore they can lend computing equipment when necessary.

1.2 Problem analysis

The primary issue to be addressed concerns hardware inventory management, which operates on two levels. Firstly, the hardware equipment is not organized properly and the location of specific components is unknown. The absence of an effective organization hinders resource accessibility and usage tracking, leading to resource misallocation and unnecessary financial impact. Secondly, the lack of a structured system for managing the borrowing process introduces another critical dimension to our issue. In the management of a large amount of hardware that can be borrowed, an accountability system is required to ensure that the equipment is properly taken care of. In addition, measures are necessary to ensure that each component is linked to the person who borrowed it and to an owner project. This mechanism ensures that if a component is not returned on time, lost or damaged, the responsible individual can be held accountable. The lack of such measures enables a disorganized practice based on trusting the borrowers intentions and not keeping track of resource damage. These practices collectively impose an unnecessary financial burden on the Pervasive Systems department, thereby limiting their ability to conduct experiments and sustain long-term investigative projects.

In summary, the problem the development team seeks to address is the absence of an organized inventory system to manage the borrowing process. This disorganized way of resource administration sparks issues ranging from lack of accountability when damaging an item to financial burdens.

1.3 Stakeholders

To provide a better overview of the context, stakeholders will be explained in this section.

1.3.1 Overview

The primary stakeholders of the Inventory Management System are the staff of the Pervasive Systems Research Group. They will benefit the most from this application, due to the system's purpose to replace their manual labor in keeping track of resources. Secondly, registering in the application is available for all people affiliated with University of Twente. Consequently, they can borrow hardware as long as they have a supervisor from staff. Thus, stakeholders can be divided into borrowers, administrators, and supervisors. This separation is optimal for requirements specification and will be presented in the appropriate section.

1.3.2 Classification

Direct Stakeholders

The following direct stakeholders have been identified as closely related to the problem of managing hardware items. These individuals or groups are directly affected by the issue and will experience the greatest effects of any subsequent changes.

- Pervasive Systems administrators
- PhD Candidates from Pervasive Systems
- People affiliated with University of Twente that want to borrow hardware equipment

Indirect Stakeholders

The following indirect stakeholders have been identified as involved in the issue at hand. Although they are not the main focus of this research, they are nonetheless considered to be affected by it.

- Companies that vendor hardware items
- Maintainers of the hardware items
- Financial administrators

1.3.3 Coordination with Client

To ensure a seamless development process of the web application, weekly meetings are held with the customer. They encourage constant interaction between the development team and the client through questions and demonstrations of progress. This approach seeks to ensure a shared understanding of the system between stakeholders and the project team. The correspondence takes place via Outlook, and the meetings take place interchangeably in the Pervasive Systems Laboratory and on Microsoft Teams. The nature of the meeting depends solely on the current stage of development. Meetings can be categorized according to their purpose as follows: requirement gathering, prototype presentation, usability testing, progress meetings with feedback sessions, and final product evaluation.

1.4 Literature review

The state of the art is represented by an Excel file that stores the name and type of hardware and the amount. The file contains information such as the item name, vendor, stock location, item description, and stock quantity. Although this sheet has provided an opportunity to monitor the number of items and their details, it was deemed to be unmaintainable in the long term. This approach serves as a temporary measure, pending the development of a more comprehensive solution. It presents limitations in terms of scalability, maintainability, and usability. Although it could be adapted to handle the borrowing process, it would have significant time costs and less robustness than a technology-based solution. Currently, the Excel file does not associate hardware equipment with any individual, resulting in a lack of accountability. These are some limitations that shall be addressed by the solution presented in this document.

Research into existing inventory systems has illustrated that numerous existing systems allow users to manage and maintain products. These existing products are mainly aimed at being used by large and small businesses alike. One such example is named ShipHero. This is a warehouse management system that manages stock movement and tracks inventory levels (ShipHero, 2025). It has a dashboard to monitor progress and there are alerts when an item in stock reaches a certain amount to help replenish the stocks (ShipHero, 2025). There are also reports and activity logs. This system is used by businesses to manage items in warehouses.

Another example is a cloud-based business management solution named NetSuite ERP. Among many functionalities, NetSuite provides an opportunity to manage inventory (Cadran, 2025). This feature enables users to view records and activity concerning a particular order. It also allows for company-wide inventory visibility, which makes inventory management more cost-effective (Cadran, 2025). Similarly to ShipHero, inventory levels are tracked and insights are provided accordingly.

1.5 Inventory Management System

The solution presented by the development team is an entire inventory management system that includes a web application with a database connection. The already existing inventory systems that were researched, such as the above-mentioned systems, have been mainly directed at businesses, and thus assisted users in maintaining their warehouse items. Despite the differences in purpose between the researched inventory systems and the Inventory System designed for Pervasive Systems, certain insights and inspirations can still be derived from these technologies. For example, the dashboard, which is used to display the stock, would also be useful for this system. The ability to see the items available for lending in one place was a functionality that the project group had planned to implement, and this has reaffirmed that idea. On the other hand, displaying alerts when the items are at low amounts would be irrelevant to this project as this project aims to trace the lending process of the items which will be inevitable returned to the stocks, contrary to the researched inventory systems which were responsible for tracing sold items which indeed do need to be replaced in the stocks.

Chapter 2

Requirements gathering

This chapter encapsulates the approach and results of the requirements gathering phase in the development process.

2.1 Methodology

To determine the system requirements, several requirement gathering meetings were organized with the stakeholders. During the first requirement gathering meeting, the client has illustrated the current electronics lending process and its challenges were discussed with the project group. This was done via a semi-structured interview, a combination of prepared and follow-up questions. The interview was conducted with the objective of understanding the entire context of the problem while ensuring alignment between user expectations and needs. Techniques such as 'thinking-out-loud' can be used if deemed necessary, in which users are asked to try out, and verbalize their thoughts as they are navigating the interface. The qualitative data obtained from customer meetings are subsequently analyzed using thematic analysis techniques, such as the use of affinity diagrams.

2.2 User Requirements

This section illustrates the requirements in the form of user stories. The stakeholders are categorized into three separate groups to help distinguish the needs of each. These groups are the borrower, the supervisor, and the administrator, and represent the role of a certain user while using the system. Administrators are some members of the PS Research Group who will manage the system and require an overview of the entire system, including all users and hardware items. The supervisor can be any current member of the PS Research Group, including PHD students. When using the system as a supervisor, users are responsible for the people they are supervising and, therefore, will have the need to monitor their activities. The borrower can be anyone who uses the system to borrow hardware items; this includes students from the UT, PS staff members, and UT staff not related to the PS Research Group. When using the system as a borrower, they will only focus on borrowing items and managing their borrowed items, without any interest in other users. As a result of this categorization of users, certain stakeholders may belong to different groups at different moments in time, depending on how they interact with the system at that moment. This leads to specific needs based on their role and goals while using the system.

2.2.1 Borrower

- As a borrower, I want to borrow items from the PS inventory so that I can use them for my project.
- As a borrower, I want to see what items are available to borrow so I know if the items I need are available.
- As a borrower, I want to easily find the items I need so that I can easily borrow them.
- As a borrower, I want to choose when I return the borrowed items so that I can use them until I finish my project.
- As a borrower, I want to know when I need to return my borrowed items so that I don't forget to return my borrowed items.
- As a borrower, I want to receive a replacement when the items are damaged so that I can continue with my projects without delays.
- As a borrower, I want my personal information to be secure against unauthorized access so that my privacy is protected.

2.2.2 Supervisor

- As a supervisor, I want students to be able to borrow items from the inventory so that they can use them for their projects.
- As a supervisor, I want to know what my supervised students borrow from the inventory so that I can ensure appropriate use.

2.2.3 Administrator

- As an administrator, I want to add and remove items from the inventory to have a clear overview of its current state.
- As an administrator, I want to see which items are currently borrowed and by whom to manage the inventory efficiently.
- As an administrator, I want to ensure that personal data is protected from security threats so that the system remains safe and secure.

2.3 System Requirements

2.3.1 Functional Requirements

Must Have

- The system must have a login system to authenticate users securely.
- The system must support three roles: administrator, employee, and guest.
- The system must allow guests to be linked to an employee.

- The system must enable employees to monitor guest activities and requests.
- The system must have a deny/approval system that allows employees to approve or deny borrowing requests from their supervised guests.
- The system must allow users to search the inventory using keywords and sorting options.
- The system must provide detailed information on the location of each item, including the status of the item (borrowed or available) and in which room it is located.
- The system must allow users to borrow and return items. The system must send a reminder notification one day before the return deadline via email.
- The system must provide administrators with an overview of the overdue deadlines for return along with the borrower details.
- The system must enable administrators to add items that could have both properties new to the system and already recognized properties.

Should Have

- The system should allow users to view their borrowed items on a separate page.
- The system should allow users to report damaged items, providing the option to submit details about the damage.
- The system should allow administrators to remove items from the database.
- The system should allow employees to add their projects to the system so that items can belong to projects.

Could Have

- The system could keep track of the items that have been ordered to indicate their upcoming availability for borrowing.
- The system could track the loan history of an item, including borrower information, return dates, and the condition of the item.
- The system could provide additional information to the borrowers about their borrowed items, including the travel limit and condition of the items.
- The system could allow the user to get a replacement if the borrowed hardware item is damaged.
- The system could contain QR codes in storage rooms that bring users to the website when they borrow items.
- The system could contain barcodes on the items, which the user can scan to immediately look for that item.
- The system could allow users to locate items based on categories and labels.
- The system could contain the already existing items in the PS inventory.

Won't Have

- The system will not allow users to reserve items in advance.

- The system will not allow multiple users to borrow the same item simultaneously, to borrow it as a group.
- The system will not automatically track each item, to provide its location in real-time.

2.3.2 Non-Functional Requirements

Must Have

- The system must ensure users can view or change data according to their authorized roles.
- The system must allow administrators to add or remove items from the inventory in no more than five clicks.
- The system must be compatible with desktop and mobile devices across various platforms.

Should Have

- The system should send reminder notifications about upcoming return deadlines for borrowed items, at predefined intervals, via email.
- The system should use a fitting color scheme based on the colors of the PS logo.
- The system should have a dark mode.
- The system should be protected against XSS attacks.
- The system should store up to 10,000 borrowing transactions without affecting performance.
- The system should enable new users to make their first borrowing request in 5 minutes without guidance.
- The system should have at least 90% test coverage in all modules.

Could Have

- The system could attach calendar notifications to reminder emails.
- The system could allow users to log in using a One Time Password.
- The system could run as a docker container.
- The system could automatically perform database backups every 24 hours, to ensure that data can be recovered in the event of a failure.

Won't Have

- The system will not allow users to log in using their University of Twente Microsoft account.
- The system will not support non-account functionalities; users need to log in to use the system.
- The system will not be accessible to users outside the University of Twente.
- The system will not include accommodations or features specifically designed to support users with visual impairments or disabilities.

2.4 Requirements Traceability Matrix

ID	Requirement
UR01	As a borrower, I want to borrow items from the PS inventory so that I can use them for my project.
UR02	As a borrower, I want to see what items are available to borrow so I know if the items I need are available.
UR03	As a borrower, I want to easily find the items I need so that I can easily borrow them.
UR04	As a borrower, I want to choose when I return the borrowed items so that I can use them until I finish my project.
UR05	As a borrower, I want to know when I need to return my borrowed items so that I don't forget to return my borrowed items.
UR06	As a borrower, I want to receive a replacement when the items are damaged so that I can continue with my projects without delays.
UR07	As a borrower, I want my personal information to be secure against unauthorized access so that my privacy is protected.
UR08	As a supervisor, I want students to be able to borrow items from the inventory so that they can use them for their projects.
UR09	As a supervisor, I want to know what my supervised students borrow from the inventory so that I can ensure appropriate use.
UR10	As an administrator, I want to add and remove items from the inventory to have a clear overview of its current state.
UR11	As an administrator, I want to see which items are currently borrowed and by whom to manage the inventory efficiently.
UR12	As an administrator, I want to ensure that personal data is protected from security threats so that the system remains safe and secure.

ID	Requirement
FR01	The system must have a login system to authenticate users securely.
FR02	The system must support three roles: administrator, employee, and guest.
FR03	The system must allow guests to be linked to an employee.
FR04	The system must enable employees to monitor the guests' activities and requests.
FR05	The system must have a deny/approval system that allows employees to approve or deny borrowing requests of their supervised guests.
FR06	The system must allow users to search the inventory using keywords and sorting options.
FR07	The system must provide detailed information on the location of each item, including the status of the item (borrowed or available) and in which room it is located.
FR08	The system must allow users to borrow and return items.
FR09	The system must send a reminder notification one day before the return deadline via email.
FR10	The system must provide administrators with an overview of the overdue deadlines for return along with the borrower details.
FR11	The system must enable administrators to add items that could have both properties new to the system and already recognized properties.
FR12	The system should allow users to view their borrowed items on a separate page.
FR13	The system should allow users to report damaged items, providing the option to submit details about the damage.
FR14	The system should allow administrators to remove items from the database.

ID	Requirement
FR15	The system should allow employees to add their projects to the system so that items can belong to projects.
FR16	The system could keep track of the items that have been ordered, to indicate their upcoming availability for borrowing.
FR17	The system could track the lending history of an item, including borrower information, return dates, and condition of the item.
FR18	The system could provide additional information to borrowers about their borrowed items, including the travel limit and condition of the items.
FR19	The system could allow the user to get a replacement if the hardware item borrowed is damaged.
FR20	The system could contain QR codes in the storage rooms that bring users to the website when they borrow items.
FR21	The system could contain barcodes on the items, which the user can scan to immediately search for that item.
FR22	The system could allow users to locate items based on categories and labels.
FR23	The system could contain already existing items in the PS inventory.
FR24	The system will not allow users to reserve items in advance.
FR25	The system will not allow multiple users to borrow the same item simultaneously, to borrow it as a group.
FR26	The system will not automatically track each item, to provide its location in real-time.
NFR01	The system must ensure that users can view or change data according to their authorized roles.

ID	Requirement
NFR02	The system must allow administrators to add or remove items from the inventory in no more than 5 clicks.
NFR03	The system must be compatible with desktop and mobile devices on various platforms.
NFR04	The system should send reminder notifications about upcoming return deadlines for borrowed items, at predefined intervals, via email.
NFR05	The system should use a fitting color scheme, based on the colors of the PS logo.
NFR06	The system should have a dark mode.
NFR07	The system should be protected against XSS attacks.
NFR08	The system should store up to 10,000 borrowing transactions without affecting performance.
NFR09	The system should enable new users to make their first borrowing request within 5 minutes without guidance.
NFR10	The system should have at least 90% test coverage across all modules.
NFR11	The system could attach calendar notifications to reminder emails.
NFR12	The system could allow users to log in using a One Time Password.
NFR13	The system could run as a docker container.
NFR14	The system could automatically perform database backups every 24 hours to ensure data can be recovered in case of a failure.
NFR15	The system will not allow users to log in using their UT Microsoft Account.

ID	Requirement
NFR16	The system will not support non-account functionalities, users need to log in to use the system.
NFR17	The system will not be accessible to users outside the University of Twente.
NFR18	The system will not include accommodations or features specifically designed to support users with visual impairments or disabilities.

Table 2.1: Requirements Traceability Matrix

Chapter 3

Architectural Design

3.1 Introduction

The central focus of this chapter is the architectural design of the Inventory Management System referred to as IMS, and capture the design choices along the process. This chapter seeks to give insight into the approach adopted by the development team to model both the software system and the environment in which it operates.

3.2 Global Design Choices

In this section of the document, design choices will be explained in detail, along with their underlying motivation. Before delving into the design decisions made during the development of the Inventory Management System, it is essential to first re-state the primary purpose of the software system being modeled. The Inventory Management System for Pervasive Systems Laboratory is aimed at solving problems in the monitoring, managing, and borrowing of hardware components. Furthermore, the system must encapsulate best practices in terms of scalability, security, and maintainability. The client emphasized these guiding principles as crucial, thereby they became the fundamental focus areas in the development of the Inventory Management System.

3.2.1 Overall Architecture

A web application with a database connection is sufficient to fulfill the purpose of storing and managing hardware components. The electronic components owned by Pervasive Systems are persistently stored in a centralized database, facilitating their real-time monitoring and effective administrative control. Moreover, the possibility of hosting the website within the University of Twente domain supports the viability and robustness of a solution involving a web-based application. Pervasive Systems research group expressed interest in having the Inventory web server deployed on their Raspberry PI, in a Docker container. This approach was deemed optimal as it ensures portability and flexibility.

The server is built in the structure of a layered monolith, and each layer provides specific services which will be further explained in following chapters. To gain a better overview of the system's architecture, this section will expand upon the high-level components of the application, along with their interaction. For the scope of this section, the classification should be as follows: the client layer, the backend application layer, and the database.

The client contains the user interface and is tailored to support the needs of three types of user, namely administrator, employee, and guest.

The application and business logic consists mainly of the handling of item storage, loan process, reporting of damage, retrieving items by filters. The IMS represents the portal for the employees of Pervasive System to borrow hardware items. The system ensures that products are linked to the person who borrowed them, enforcing accountability. The backend application is built using the Model-View-Controller design pattern which will be expanded upon in the following section, and explained in much more detail in chapter 5.

The database is a main component within the Inventory Management System due to its purpose of storing and retrieving information about users, hardware items, and borrowings of such components.

3.2.2 Design Choices in Modeling

This section focuses on the rationale behind the design choices when developing the Pervasive Systems Inventory Management System. Due to the complex nature of the environment in which it operates, certain abstractions were necessary.

Users of the systems

Firstly, certain limitations needed to be defined in terms of users in the IMS, the development team modeled the user with the assumption of their affiliation with the University of Twente. This assumption is informed by the requirements provided by our stakeholders and the intended scope of the system, restricted to users within the university community. This limitation is also needed to ensure identification with the student/employee number and card provided by the University of Twente. The direct stakeholders are part of Pervasive Systems, therefore they can identify with their employee number, and in projects they collaborate with students or other departments of the University of Twente, thus the hardware items shall only be borrowed by them.

Another design choice in the modeling of users is the different roles a use can have in the system. As already mentioned in the requirements chapter, the inventory management system works with 3 different roles for the users, namely administrator, employee, and guest. Guests are the most basic types of user and are meant for users who are not part of the PS research group, such as students of the University of Twente. Guests can view the inventory, but cannot add or alter items in the system, and as they do not have access to the storage rooms, they will need an employee or administrator to be their supervisor and accept requests for them to be able to borrow items. The other group of users are employees, who are users related to the PS research group. They can directly access the storage rooms, and therefore are able to borrow items in the system whenever they want. Employees are also able to add new items and projects to the system. The group of users are administrators, who are employees that have been tasked to manage this system and hardware lending in PS. Therefore, administrators have some additional permissions, such as having an overview of all the users and borrowed item, so that they can manage the system. Table 3.1 provides a visual overview of the different permissions for each of the roles.

Guest	Employee	Administrator
View inventory	View inventory	View inventory
Request borrowing	Borrow items	Borrow items
	Supervise guests	Supervise guests
	Add projects and items	Manage projects, items, categories, and users

Table 3.1: Permissions per user role.

Hardware Items

When it comes to hardware items, systematic management was needed to ensure scalability in the database, so that it can handle the diversity of possible hardware types and specifications. Furthermore, the database shall be adapt to the administrators adding completely new hardware with different technical attributes (voltage, amperage, model, type, size in GB). To achieve this, a system of categories and variables in categories was developed. To illustrate this, consider a category "Headphones" with variables connection, frequency response, charging time in hours, and noise-canceling. The design needs to adapt to these different types of variables and allow users to create new ones. This approach ensures the scalability and flexibility of the Inventory Management System in the sense that it can model any type of hardware item.

A second aspect to take into account when modeling hardware components is to what extent some need to be tracked. This implies that for some items it is essential to identify each piece distinctly. Such hardware components can be microcontrollers, devices, peripherals, or any type of hardware whose state is crucial to monitor. It is of high importance that in the event of a damaged microcontroller, users can accurately identify the specific unit affected. This measure prevents the unintended use and borrowing of a damaged item. Another category of hardware components is the one that is managed the bulk; therefore, it is not required for each separate item to be identified, the system shall only track the number of items available. Examples of such items include wires, resistors, and capacitors. Finally, there are the types of items that do not require a record of number or state. These items can be screws, solder paste, zip ties, staplers, or other items that are uncountable or that are sold in large quantities. The development team has chosen to model the collections of items in a way to encapsulate all these possible types, thus they can be unique, bulk, infinite. Unique items have their own unique identifier, thereby allowing their state to be monitored, while for bulk items only their count is managed in the system. This design choice is intended to enhance the system's scalability such that it can accommodate various types of items without the need to maintain unnecessary counts or identifiers.

Borrowing

Given the complex nature of the borrowing process, certain aspects are abstracted. For example, the system cannot know if the hardware item is not in the position or location indicated by the user interface. The Inventory Management System does not provide GPS

tracking of items or anything of the sort that would allow dynamic positioning of each item. However, the system generates QR codes for each item that can be downloaded and put into the physical environment, enabling people to just go in the room, scan the QR code and borrow the item. The application allows users to borrow items from the website itself. This design choice was made to ensure the convenience of the user and a seamless process of borrowing hardware items.

3.2.3 Programming Languages and Frameworks

This section will cover the choice of programming languages and frameworks in the development of the IMS.

Java Spring Boot

For the business logic, Spring Boot was deemed the most suitable option due to its extensive ecosystem and libraries, convenience of use and scalability. This framework presents significant advantages in terms of rapid development of a web application, scalability, and performance by successfully handling highly concurrent environments. Spring Boot while being in the domain of expertise of the development team, it shows significant performance in the stakeholders' points of interest, namely scalability and security. Given the limited scale of the research group, it was deemed unnecessary to aim for better performances than Spring Boot offers, as the system is not expected to manage intensive workloads.

Hibernate and Java Persistence API

Java Persistence API is a specification often used when mapping Java objects to database tables. This is best illustrated using an example, there needs to be a mapping among the user model in the Java program, and the user table inside the database, to ensure that CRUD operations can be performed. Hibernate is an implementation of this specification encapsulating its improved features and it is used in the development of the Inventory Management System made for Pervasive Systems Research Group.

MariaDB

MariaDB is an open-source database management server with no commercial version barrier. It is comparable to MySQL in performance and it presents additional and improved features. MySQL is widely used and known and it is enough for the load and data that needs to be stored within the Inventory Management System. Although popular options such as PostgreSQL provide greater performance and more features, MariaDB offers greater flexibility and fault tolerance at scale. The additional features of PostgreSQL are not within the scope of the Inventory Management System for Pervasive Systems Laboratory. Moreover, greater speed is not necessary in this scope at the expense of robustness and stability. Flexibility of data is of utmost importance in the problem context because there are many variables and technical specifications that must be stored. The last and main reason for choosing MariaDB instead of a more powerful alternative is that it requires less memory and CPU usage. As mentioned above, the web application will be deployed on Raspberry PI, thus there will be less resources available which makes MariaDB a suitable option.

React and JavaScript

React was the chosen JavaScript library for this project. It helps write HTML-like code (JSX) within JavaScript. Due to its component-based architecture, there are reusable components which reduce the amount of duplicate code. Thus, the structure of the project becomes more maintainable and flexible. Additionally, the built in functions such as hooks and state management help with managing data in an intuitive manner.

Bootstrap

For efficiency in user interface implementation Bootstrap was used. As a CSS framework, Bootstrap provides numerous pre-designed and customisable user interface components. The built-in grid system enables the application to be responsive in various screen sizes. These are few of the reasons Bootstrap is used the development of the Inventory Management System for Pervasive Systems.

3.2.4 Design Patterns

Model-View-Controller

The overall architecture of the web application is based on the Model-View-Controller pattern. It separates concerns into the following categories: model which encapsulates the data and business logic, view which consists of the user interface and controller which handles the HTTP requests. This approach separates the user interaction from the logic of the system, thereby enabling for flexibility in changing the different components without affecting the rest. This design pattern also allows for reuse of the model across different interfaces or environments, while ensuring scalability. The Model-View-Controller design pattern was used due to its numerous advantages including its promotion of clean architecture and maintainable code. It is widely regarded as an optimal practice for enhancing quality of code and facilitating structured software development.

Repository Pattern

Due to the adoption of a layered system architecture, data access layer is decoupled from other components and is managed following the principles of the Repository Pattern. This pattern consists of abstracting the persistence logic and dealing with SQL queries. It facilitates the operations on data, while allowing for customization. As a result, this pattern was chosen to simplify data access and mitigate the vulnerability for SQL injections.

Service Layer Pattern

The Service layer handles the business logic of the application. Separating it from the control and data access layer is beneficial for ensuring the Single Concern Principle. This states that each component of the system shall do a single task, and separating the roles is an effect of the principle. The Service layer pattern enhances the maintainability and flexibility of the software, therefore it is a practice that the development team employed for building the Inventory Management System.

Singleton Pattern

This pattern is mainly embedded within Spring Boot, each Spring bean, service, configuration is a singleton. An effect of this feature is that it is not required for the developers to manage objects or instances lifecycle.

Dependency Injection

The dependency injection is another design pattern embedded in Spring Boot that promotes loose coupling, thereby facilitating unit testing or mocking. One advantage of employing this pattern is that there is no need to manually create objects in classes that need them, Spring injects them automatically.

Proxy Pattern

A proxy is a wrapper around a real object that controls access to the object and that can add extra behavior before or after calling the real method. This pattern is a foundation for the way Spring operates. In this implementation one can find the pattern in the logging system that will be explained in the following section.

3.2.5 Logging

Logging is a crucial aspect of the server in order to determine reasons for system crashes and faults. Dealing with a web server, the logging is part of the main configuration and it is initialized using a factory method. It will log information regarding the server connection, the initiation of the container, the configuration, the servlet engine, and the port in which the server starts along many others. The development team has decided to implement logging within the Inventory Management System to enable for easier debugging and troubleshooting. Furthermore, the feature can by utilized for performance tracking and security auditing.

3.2.6 Error Handling

This section comprises of the manner in which the IMS handles the runtime errors. It is inevitable that the system will run into errors, either caused by internal or user-related factors. The system faults and errors should, however, not impede the availability of the server, therefore it should effectively recover from them without crashing. To illustrate this, the section will discuss error handling on two different planes, visibility and recoverability. Firstly, the errors are logged into the system, so they are visible immediately to the developers. In case the error is caused by user input, the system notifies the user of the issue providing an alternative action or approach. Secondly, the system recovers from failed attempts to change data or from bad input from the user. The fatal problems are mostly related to data integrity violations or lower level faults.

3.2.7 Scalability

One of the main points of interest of the Pervasive Systems is that their system is highly scalable. They have multiple types of hardware and the system should accommodate the possibility of modeling these unknown hardware components. We achieve this in our database design, being able to add any type if item with any kind of specifications. In

this document, technical specifications of the hardware components will be referred to as variables for an item collection. The scalability lays in the tools and frameworks used to develop the system and is embedded in the architecture itself. The exact design choices that lead to a more scalable system include using Spring Boot, including variables as technical specifications and the way they are modeled in the database.

3.2.8 Portability

Portability refers to the ease of setting up the application across different devices and environments. This is achieved by packaging the code, dependencies, runtime, configuration and tools into one Docker image. This enables platform independence by abstracting away host differences and running the application in a predictable way, regardless of where it is deployed.

3.3 Preliminary Design

This section shows the initial design of the Inventory Management System which has been refined in each stage of development. The diagrams present the initial vision of the development team in the IMS design and serve as a baseline of what the system will become throughout the process. The diagrams are made using UML, which is the standard for software development.

3.3.1 Structural Modeling

Structural modeling mainly addresses the architecture of the system. It showcases classes and objects, alongside the vision for the database.

Class Diagram

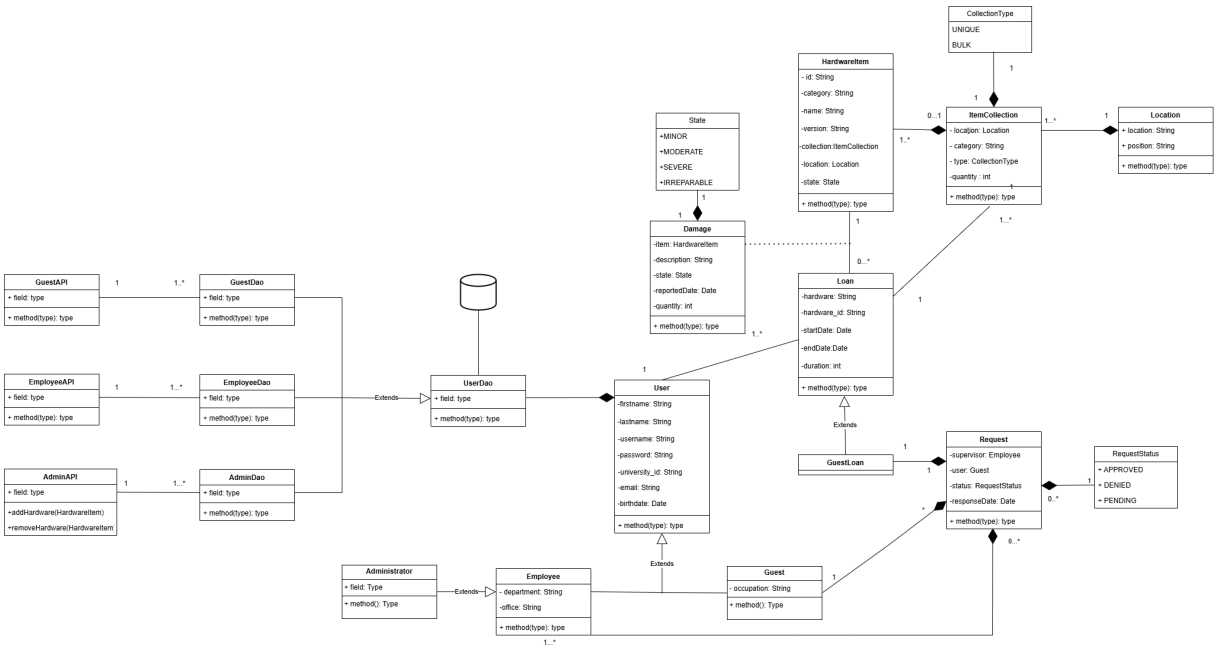


Figure 3.1: Initial Class Diagram of the Inventory Management System

The diagram above (figure 3.1) illustrates the first concept of an Inventory Management System for Pervasive Systems Laboratory. At this point in time, it was not possible to anticipate all the practical aspects and considerations when developing the IMS, therefore, the diagram is not consistent with the resulting system. The same holds for the database schema below (figure 3.2), which has seen multiple iterations of changes and improvements.

Database Schema

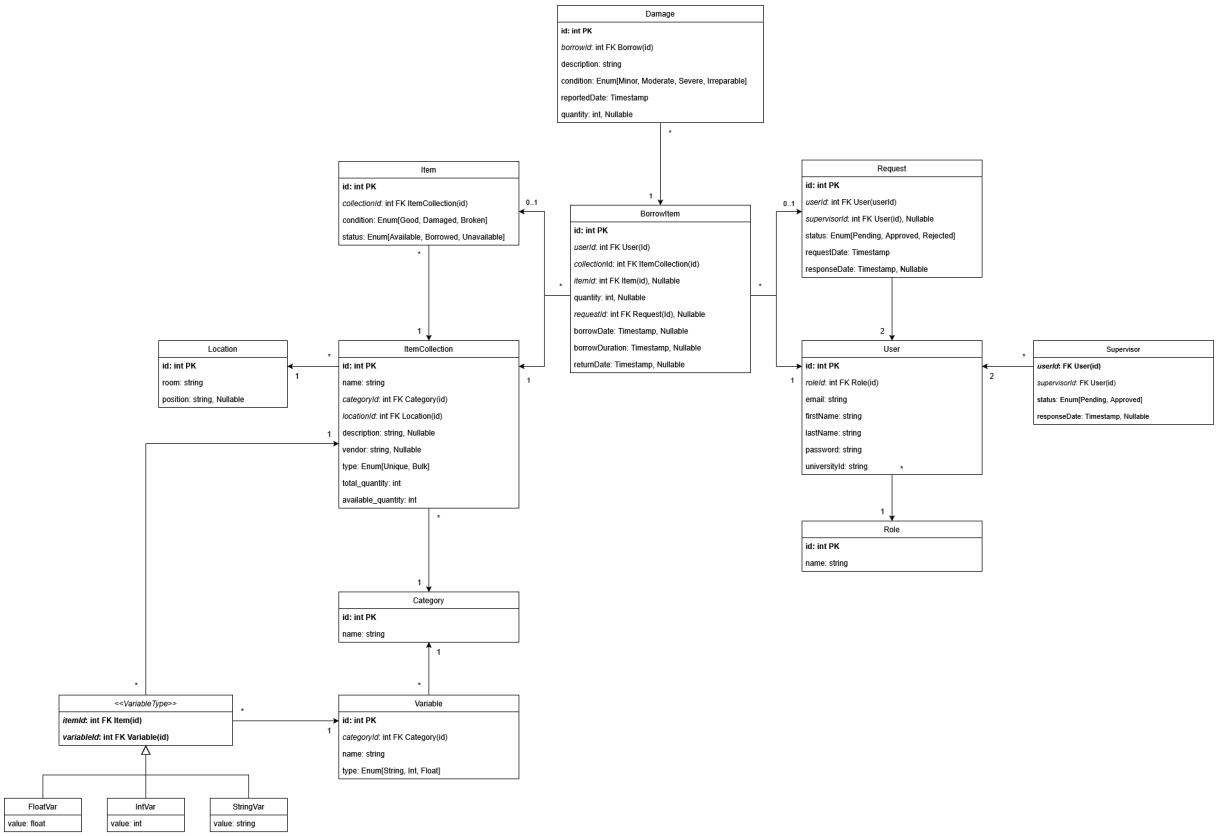


Figure 3.2: Initial Database Schema of the Inventory Management System

3.4 Final Design

3.4.1 Structural Modeling

Component Diagram

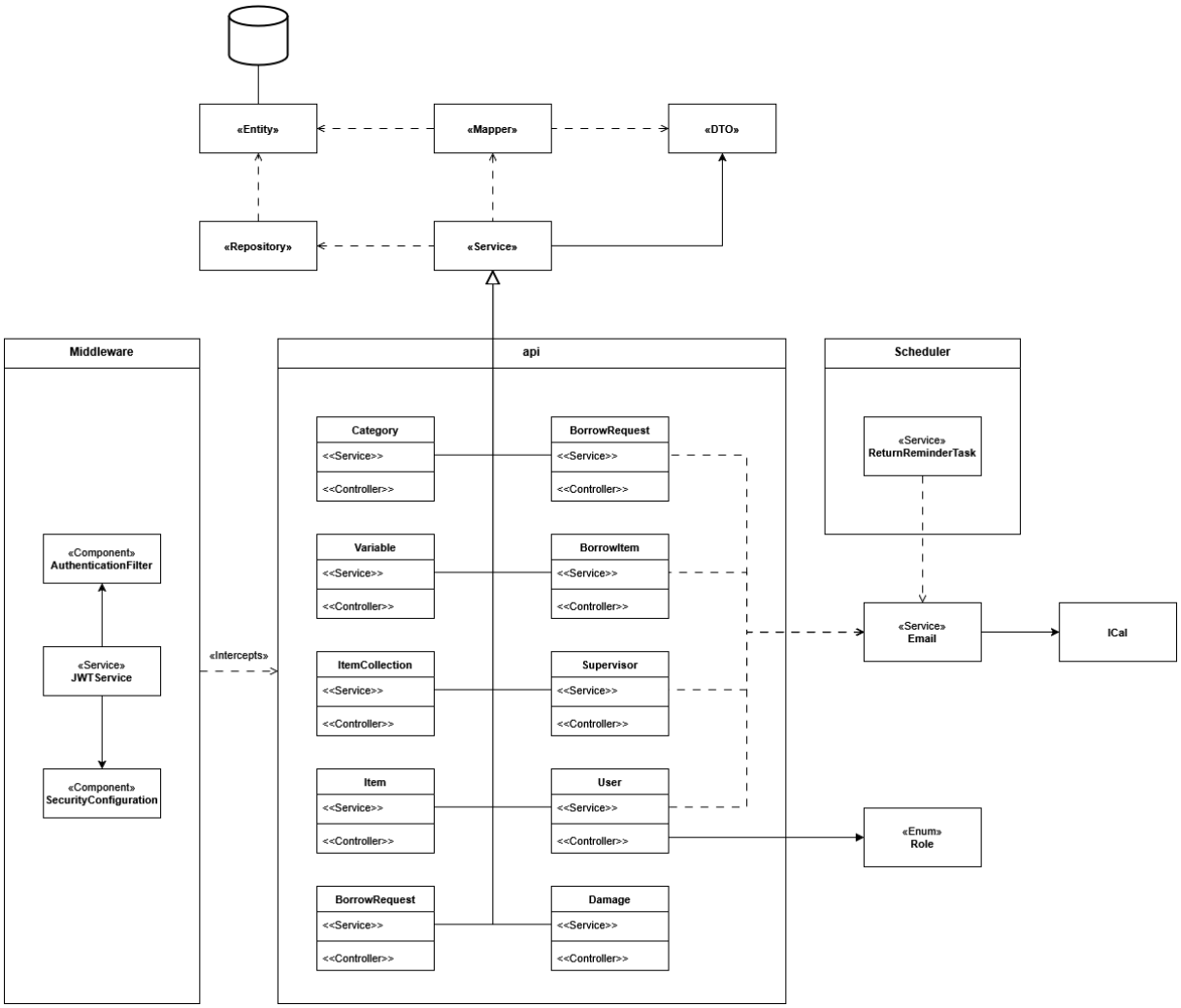


Figure 3.3: Final System Diagram of the Inventory Management System

The figure above (figure 3.3) shows the final class diagram, and in the diagram below (figure 3.4) the final database schema can be found. These final diagrams show how the system ended up looking, and provide some more detail and additional components compared to the initial versions, due to troubles when implementing the system and additional requirements that were added during development.

Database Schema

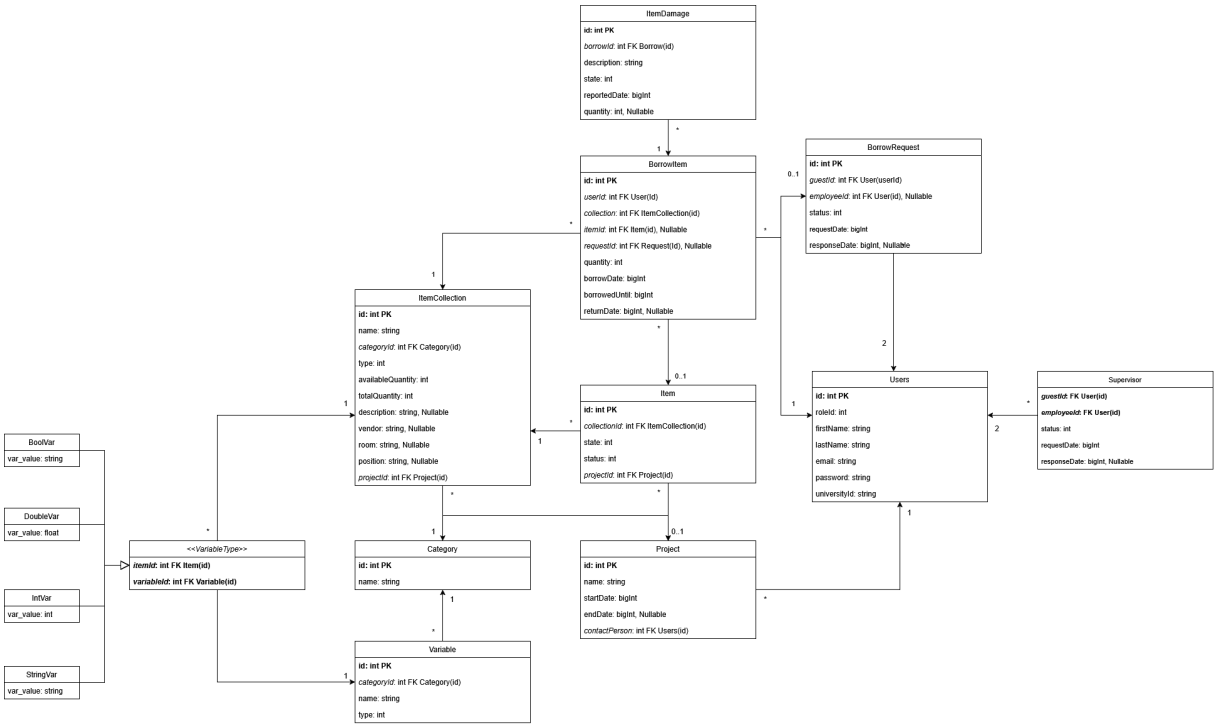


Figure 3.4: Final Database Schema of the Inventory Management System

3.4.2 Behavioral Modeling

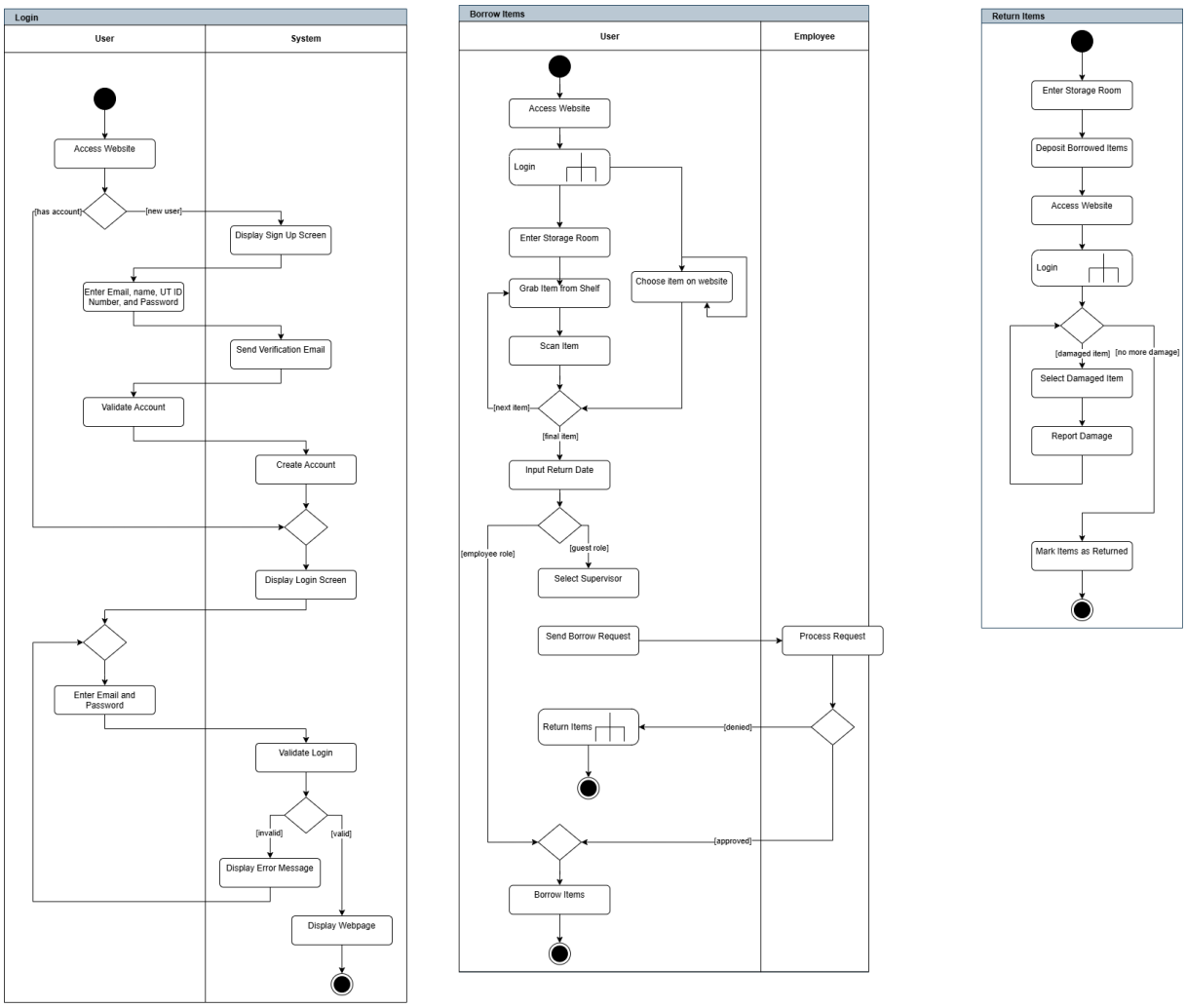
This section provides the use case and activity diagram, which show the behavioral modeling of the Inventory Management System.

Use Case Diagram

Figure 3.5: Use Case Diagram for the Inventory Management System

The use case diagram above (figure 3.5) and the activity diagram below (figure 3.6) are relevant for both the initial design and the final design, as they are only slightly altered to match the final implementation of the system.

Activity Diagram



Sources Login:
https://www.researchgate.net/figure/Activity-Diagram-for-Sign-Up-and-Log-in_fig3_277014901
https://creately.com/diagram/example/hcjhgsl91/activity-diagram-for-user-login

Figure 3.6: Activity Diagrams for Login, Borrowing, and Return Processes

Chapter 4

Mock-up and prototype

In this chapter the design choices made for the mock-up and prototype will be laid out. Screenshots of all mock-up pages are provided in Appendix A.

4.1 Preparation and scope

Before a start was made on the mock-up, the libraries that are used in the front-end needed to be decided. This was done to ensure that the mock-up is an accurate representation of the final product. Section 3.2.3 already goes over more details of all libraries that were used, for this section only the React and Bootstrap. React is used for structuring the pages into different components, so for the visual look of the pages this library can be disregarded. However, Bootstrap does play a big role, as it provides visual elements that can be used.

For creating the mock-up, the decision was made to use Figma as it allows for community asset packs which can be used when creating drafts. One of these asset packs is the Bootstrap 5 UI Kit by Jitu Chauhan, which provides recreated components of the bootstrap elements.

During the requirement phase, there were multiple non-functional requirements listed that focused on how the interface should look, and what needs to be possible. For the mock-up (and later the prototype), we wanted to put the focus on how the user goes through the system in order to borrow an item, as the borrowing process of the system will be used the most. Due to the restriction of the flow we wanted to demonstrate, a lot of functional and non-functional requirements will not be represented within the mock-up. However, we did put a focus on NFR03, for which we were given the Pervasive Systems logo and colors.

4.2 Designing the prototype

For designing the mock-up, a color scheme was established. This was done in a way that it would replace the bootstrap primary, secondary, light, and dark colors, allowing the color scheme to be applied easily to the front-end in the future. The primary is the same color as the text on the PS logo, with the light and dark colors being a lighter and darker variant, respectively, of the primary color.

During the creation of the mock-up pages, the aim was to keep the information presented on pages to a minimum, to prevent overwhelming the users. This is especially true for the home page (Figure A.2), where the user needs to be able to quickly choose the

actions they want to perform (borrowing or returning items).

Furthermore, some focus was put on things that required feedback. This is the reason for the decision to add the filtering of items to the mock-up (Figure A.5), where easy use is a priority to allow users to quickly find the items they need.

As for the functional aspect of the prototype, it is kept to a very minimum. All pages shown in Appendix A are the only versions made for the pages, the data on the pages do not change depending on input. An example of this is that all items on the item list (Figure A.4) will lead to the same page, a unique item collection, (Figure A.6), to view the shared collection (Figure A.7) it is possible to switch between the two by clicking on the information of the page.

Chapter 5

Implementation and detailed design

5.1 Application Interface

This section of the implementation will focus on the systems used for the front-end. For each of the systems created, there will be a short explanation of the system along with the reasoning behind the design choices. This section does not contain the decisions for all of the individual pages, as that would cause this section to be long-winded, although some of the more important design choices are included.

5.1.1 Scripting utilities

The scripting utilities are used for scripting. These do not create any visual elements, although they can be used to decide the content of visual elements.

ApiRequestData class

The `ApiRequestData` class is a helper used to get data from the API. Due to the nature of React, the pages need to be able to be re-rendered without spamming request unnecessarily The intention for this class is to keep track of the request progress, and once the information updates to re-render the current page. Additionally, it handles the errors for which the action performed should be the same regardless of the location where the request was issued. An example of this is the handling of an `HTTP 401` response, which causes the session to be cleared by the `ApiRequestData` class.

When creating a class, it requires information about how the requests should behave. It is possible to make a request `allowMultiple`, which, when enabled, will allow the request to be called multiple times, as long as the previous request has finished. However, if this is disabled, it will only allow the request to be achieved, unless the user refreshes the page, or the request is forced. This is very useful for creating request to gather page information, as it prevents the page information to be regathered each time the page needs to be re-rendered. However, for an action performed on the website, `allowMultiple` should be enabled, as the user can perform the same action multiple times without refreshing the page.

Date converters

There are multiple date converters provided as the API uses UNIX timestamps, however, JavaScript uses their `Date` class. To help with this conversion, four methods are created:

- `UnixToString` - Convert a UNIX timestamp into a string representation (e.g. Sat Apr 19 2025)

- `UnixToDate` - Convert a UNIX timestamp into a `Date` object.
- `DateToUnix` - Convert a `Date` object into a UNIX timestamp.
- `StringDateToUnix"` - Convert a string date gathered from a form input into a UNIX timestamp.

Dev helpers

With the way the back-end and front-end are set up, it is possible to run both separately, which has the benefit that the front-end automatically updates without the need to refresh the page each time. However, this has the downside that the front-end will run on a different port than the API. To prevent conflicts between development and production environments, there is a `getApiUrl` method provided which automatically appends `"http://localhost:8080"` in front of the given input URL in the development environment, while in production it simply returns the input.

5.1.2 Context systems

In React it is possible to use "contexts", these are inserted as normal HTML components in your code with a single value associated with them. When requesting this type context system from any child components, it will search up the HTML tree for the first context provider it encounters. It takes the value from the provider and gives it to the requesting component. In this way, it is possible to easily share data between different components in React.

Alert system

The alert system is a global context, meaning that there is only one context provider within the body that contains all the page contents. This means that the alert information is shared across the page. Using this, it is possible to have only one alert element, which can be shown with the desired content from anywhere within the page.

Bag system

This is another global system, that keeps track of the items the user has added to their bag. This system is also synced with the local storage of the user, in order to keep the bag contents even after the user logs out or refreshes the page. The bag system does not just save the ID of the object that was added, but also the name so that it does need to wait for the request to complete to show the item name when viewing the bag.

Session

The session system is always created, even if the user is not logged in, as this is another global component. It keeps track of the session information, including when the session expires. This is also what is used by the `ApiRequestData` to clear the session data whenever an `HTTP 401` is received, as it can be accessed from anywhere. Similarly to the bag system, this is also stored in the local storage to ensure a page refresh does not cause the session data to be lost.

State forms

During the development of the user interface, it became evident that form handling occasionally conflicted with React's component-based architecture and state management approach. The form contents would be reset after clicking the submit button, which was not something that could be prevented, even if the submission failed, for instance, due to the user entering invalid information in the form. To prevent this from happening, state forms were created. These are not global components, as multiple are needed. They only need to have the form components as their children. The state form keeps track of the information entered into the form within a React state, which is persistent between page re-renders (however, lost when refreshing).

The downside with this method is that all the form input, now also needs to use custom React components in order to interact with this new state form context system. These new form input has the same capabilities as normal form input, although some common properties are already set for easier use. Whenever the input within a form input is changed, it needs to report this change to the context, which also causes the page to re-render.

However, a big upside is that there is now full control over when the form gets cleared, with the added option to have an asynchronous action determine when to clear the form.

Modals

Another issue that became apparent during the development of the interface, was that using the bootstrap modals (see the bootstrap documentation) caused issues with React. With re-renders the reference to the modal would frequently be lost, meaning modals could not be closed using scripting anymore, additionally it was difficult to pass information into the modal (e.g. the item information to display). The modal was created to keep track of references that may otherwise be lost due to page re-renders making it possible to always hide the modal. Additionally, with the show method, it is possible to include page information which will be given to the modal.

5.1.3 Components

Navbar

The contents of the navigation bar are generated based on JSON information and current session. This data contains the text for each of the buttons and URL the button should redirect to. Also, additional links can be specified, to help with highlighting the correct button. An example of this is that the `/inventory` is the URL the `Inventory` button points to, however the additional links includes `/inventory/*`, which means the button is also highlighted whenever the user views any specific item (`*` indicates a wild card).

API button and form submit button

When performing actions, it is important for the user to know whether the action is still pending or has already finished. For this 2 components have been added, both of which perform in similar ways; however, one is a normal button, while the other is a form submission button. They both take an `ApiRequestData` object in their input to track

the state of a request. If the request is still ongoing, it will show `Processing...` as the button text, but if there is no ongoing request, it will show the normal button contents (the children within the HTML structure).

Dark mode toggle button

The dark mode toggle button controls the dark mode for all pages. This is also the reason the button needs to be included on all pages, regardless of whether the user is logged in. The button has two modes, "small button" and "drop-down button". The small button is an independent user interface element that solely contains an icon indicating the current status of the dark mode setting. In contrast, the drop-down button is used within a drop-down menu and also contains text. The button synchronizes with the local storage of the browser in order to make dark mode persistent for a device, and it changes whether dark mode is enabled by changing the `data-bs-theme` attribute on the body object of the page, this indicates to bootstrap which mode should be enabled (`light` or `dark`).

Tool-tip

To add additional information to some buttons or pieces of text, tool-tips can be used. Bootstrap has tool-tips build in, however similarly to Modals (discussed in section 5.1.2) the references to tool-tips are lost between page re-renders. A solution was to make our own custom tool-tip component to assist bootstrap with the references. Each tool-tip will need to be given a unique id, this is stored in a global JavaScript variable. Because the global JavaScript variables are not reset between re-renders, this means we now have a persistent way to keep the reference to the tool-tip, without the need for React states. This is important as a page will need to keep the same amount of React states between re-renders, but the amount of tool-tips may vary (e.g. a request was made at the first renderer and tool-tips were only added based on the received response during a later re-render), so using states is not an option here. Whenever the tool-tips detect an old reference is in the global variable, it will hide this tool-tip to make it disappear as it otherwise gets stuck on the screen, while also creating and storing the reference to the newly rendered tool-tip.

Hooks/Queries

In addition to the ApiRequestData class, the project includes some customized hooks that are used to make API requests. These hooks are situated within the hooks folder and were made into reusable and callable methods that make API calls, handle errors and return the retrieved information. This information is later on used in other pages. The reasoning behind separation of these callable hooks is to make code more maintainable and neat.

5.1.4 Pages

In this section, an overview of the pages will be provided. Components such as the header, navigation bar (Navbar), a link to the bag and checkout, and a small menu with an option to toggle dark mode, view account details and logout of the system are provided in and accessible by all the pages. Due to different permissions each account has, some functionalities are only visible to and accessible by employees and admins and several

exclusively by admins. Examples of these functionalities are manage pages which will be further explained.

Home

The home page is the one that the users are navigated to upon logging in. There are two buttons that help the user easily navigate to borrowed items section and view inventory page. The borrowed items section is positioned in the second half of the home page. This design choice was made to ensure that the users would have direct access to an overview of their currently borrowed items. The navigation bar that is situated on top of the page enable a smooth navigation through all the pages.

Borrowed Items

In this section, the users can view their currently borrowed items, along with item id, item name, borrowed quantity, and the required return date. By clicking on the buttons of each borrowed item, the user can perform several actions. These include returning an item as well as extending the borrow duration by a given number of dates with a limit of thirty days, and reporting an item as damaged. It should be noted that extending and damage reporting are supported only on the individual items with a unique id. Therefore, it is not supported to extend the borrow duration items without a unique identifier such as small cables and padding.

Layout

Layout helps combine navigation bar, bag, session, alert and account components. With the help of layout, these components become accessible in all the other pages. This page is also responsible for directing the user to the login pages if their session expires, ensuring secure access control throughout the website.

Inventory

In the inventory page, users can view, search for, and filter items within the inventory. With advanced filters, users may indicate the category and after stating the category, it is possible to indicate which variables of that category they want to enable in the filter. If the variables are not indicated, then the filter is applied by the selected category. An example of the advanced filters used to indicate the variables are, for category Arduino, clock speed, flash memory and digital I/O pins.

The inventory items are provided on this page, along with their availability. The view button on each item navigates to a page that includes more specifications on the chosen item. On this page, a user may see all the given details of items, their vendor, room, position, and the project they belong to. Then they can see all the instances of this item, their availability and condition (good, damaged, broken).

All users can see the barcode, availability and condition of given item. The admins can also modify collection details, add additional items, and delete items. Admins can also modify the condition of an item (e.g. setting a broken but repaired item to good), and view the borrowing history of an item. The first step of the borrowing process also

takes place in this page. All users may select from available items and add them to their bag, which they visit later on to checkout and complete their borrowing process.

Supervision

There are two interfaces for the supervision page. These are, one for the supervisors (the employee and the administrators), and the supervisee (the guest).

Supervisor (Employee and Admin) View In the supervision page of employee and admin accounts, within the supervising section, a list of the users that they supervise are provided. These consist of users that have previously requested to be supervised by the supervisor, and the supervisor has accepted their request. On this page, the supervisor can view or remove a supervisee, which removes the connection between their accounts. Additionally, on the open requests section, the supervisor receives and views supervisor requests from the guests accounts. They may accept of decline these requests. Lastly, in the borrow request section, the supervisor receives borrow requests by the users that they supervise. They may view the item that the supervisee has requested to borrow, the amount and the date that they made the request. Afterwards, they may choose to deny or accept these borrow requests.

Supervisee (Guest) View In the supervisee view of the supervision page, in the create a new supervisor request section, the guest may enter the email of their supervisor and create a supervisor request. If this request is accepted by the supervisor, then the guest becomes supervisee of that supervisor. Underneath in the your supervisors section, there is an overview of the supervisors that the guest is linked to. The guest may choose to remove a supervisor. In the open requests section, the guest may view the pending supervisor requests that they have made. Lastly, in the borrow requests section, they can view the borrow requests that they have made, along with the supervisors that they have made these requests to.

Checkout

In this page, the users see an overview of the items that they have added to their bag. The checkout is accessible by clicking on the bag located on top right corner of each page. This is the second step of the borrow procedure after adding an inventory item to their bag. Then, users are prompted to provide a borrow until date. The maximum borrow duration is three months for guest users, and six months for admin and employee users. This duration is extendable by thirty days later on. Then, by clicking borrow all items, users borrow the selected items. This direct borrow is applicable for employees and admins. As for the guest users, in order to borrow an item, they need to make a borrow request to their chosen supervisor. Then, as explained in the supervision page for the supervisors, the supervisor may accept the borrow request.

Error Page

This page receives a status and a return URL. It checks the received status and displays the corresponding message on the page. For instance, for status 404, it returns a "Not Found" status information along with a friendly message "The page you are trying to access could not be found." that the users can easily understand. At times the user may

try to access a page that does not exist, at other times they may try to access a page that they do not have enough user privileges for, and sometimes the server may have an error and does not return the requested page. Regardless of the status and the situation, this error page smoothly handles the errors and informs the user about the encountered situation. Additionally, when a return URL is provided, this page provides an option to return to the page of the given URL. And always includes an option to navigate to the home page.

Manage

The pages related to management are only accessible via users with an administrator and employee roles, and some only admin. These pages enable users to create, modify, view and remove several components of the inventory. In the following sections, they will be further illustrated.

Create Item In this page, the admins and employees can create new item collections. The collection name, description, vendor, location of the item are given to make a new item. As each item has a collection type (bulk, unique, infinite) and category, and an initial item count, these are provided by the user.

Manage Categories In the manage categories page, admins can view, search for, create and delete categories. This page is only accessible by admins. It is also possible to create or delete already-existing variables that belong to a category. These editable variables may include but are not limited to connection, dimension, model, speed, RAM and interface. This page assists admins with an intuitive navigation through category management.

Manage Projects Another page within the management pages is the Manage Projects page. This page is accessible for both employees and admins. Each project has a project name, start, and end dates. The inventory items are often linked to inventory items, and have a contact person. Within this page, users can search for, view (both an overview of all the projects, and each project details individually), delete, and create a new project.

Manage Users This page is exclusively accessible to users with a role of admin. Within this page, an overview of all the users within the system are shown. The admin can search for a user, view the currently borrowed and returned items of a user, filter through the users, and can delete a user from the system. Additionally, administrators can also change the role of a given user (admin, employee, guest) in order to increase or reduce their user privileges.

Authentication

The website supports several manners of authentication for user convenience. After signing up as a user, it is possible to login by using email and password. Then, a three-hour session is initialized. It is also possible to login by using a one time password (OTP). This password is sent to the email of the user upon request and is valid for five minutes. Therefore, if a user wishes to have access to their account via OTP, they would need to request it every time.

The system also supports password reset. After requesting a password reset on website, the user receives an email with reset instructions as well as a link. This link is active for fifteen minutes. The user can then follow this link and provide a new password for their account.

5.2 System Infrastructure

5.2.1 Presentation Layer

The presentation layer of the system consists of all API endpoints that the application interface uses to request, create, update, and delete data in the system. This layer is concerned with handling interactions between users and the system.

REST controllers are key components of the presentation layer, as they serve as entry points for HTTP requests by interpreting incoming data, invoking appropriate service layer logic, and returning responses in JSON format.

Each REST controller has its own request mapping, which is a general path to that specific controller. For instance, `/api/user` points to the user controller, which handles operations involving users.

Within each REST controller, there are multiple mappings of the HTTP methods. The HTTP methods used within the system are GET to retrieve data, POST to insert it, PUT to update existing data, and DELETE to remove it from the system.

5.2.2 Business Layer

This is the layer that consists of the application's business logic. The service layer deals with operations using instances of the data access layer, validates data received from the upper layer,and performs additional checks and operations to ensure consistency within the system.

In the Inventory Management System, the business layer encompasses all service components, specifically those annotated with @Service. Examples include UserService and CollectionService, which are integral parts of the IMS architecture. In the context of Spring Boot, a bean refers to an object that is managed by the Spring container. These service classes are instantiated, configured, and maintained by the framework to facilitate core business logic operations within the system. This is a crucial aspect in the development of the Inventory Management System due to the fact that there was no need for each of the services to be instantiated, and they could just be declared in other classes, leaving Spring Boot to handle their injection.

For a better understanding of the application, it is paramount to first gain insight into dependency injection. Some of the services are interdependent, instances of a service are used by others, and this happens automatically, being part of Spring Boot core architecture. Spring Boot scans classes annotated with @Component and its specializations, namely @Service, @Repository, and @Controller. Furthermore, it registers them as beans within the application context, and they become available for injection whenever necessary. For

example, in this application, the appropriate service is injected into each controller to deal with the logic and data operations. Moreover, requisite components are injected into each service to handle operations that fall outside the primary responsibilities of the current service.

5.2.3 Data Access Layer

The data access layer consists of all JPA repositories within the Inventory Management System. Java Persistence API is mainly characterized by its lightness and its slimness. One of the main features of the JPA is the specification of the object- relational mapping by the Java annotations directly in the persistence object. (Böck, 2011) The separation of concerns is a paramount principle in the development of this application, therefore this layer is mainly focused on accessing the data.

JPA integrated with Hibernate provides a framework for developers and predefined functions within the repository that find objects in the database based on certain fields. It automatically generates SQL queries for based on certain fields declared as entities in the database. More about Hibernate entities will be presented in the following section.

5.2.4 Model Layer

The model layer encapsulates the objects modeled within the system, mainly the ones found in the database, therefore this layer is focused on representing data. When using Hibernate, the models in the context of Inventory Management Systems are the entities, namely classes annotated with @Entity. Hibernate directly maps entities to the tables in the database.

5.2.5 Data Transfer Object Layer

For security purposes, the usage of entities within the services and in controller returns is not recommended. Such practice may reveal parts of the database structure, that makes the web server vulnerable to injections. The structure of database should remain unknown to the general public as it represents a fundamental component of the system. To avoid sending entities as responses and managing them directly into the services, a new layer of data representation was adopted, namely the data transfer object layer. This part of the software contains alternative models to be sent to the upper layers of the application, to not disclose the database structure to other levels of the IMS. It is considered good practice to use only the data transfer objects at the controller level, thus the Inventory Management System conforms to this principle and is implemented accordingly.

5.2.6 Mapper Layer

This layer represents the intermediate between the model and data transfer object layers, achieved by mapping each entity to its DTO and vice versa. The application needs a mapping among data transfer objects received from the client at the controller level and the entities which are needed for data processing and alterations. For data migration between the data access layer and the presentation layer, this mapper layer is crucial.

Spring Boot handles the actual implementation of the mappers. It is enough for the

development team to provide an interface of the transformations required and the naming between entities and data transfer objects to be consistent for Spring to be able to generate implementations in which each field of the transfer object is mapped to the field having the same signature inside the entity. Moreover, mappers can use other mappers to transform object fields into entities.

5.3 Components

This section aims to provide future developers and maintainers of the Inventory Management System of the Pervasive Systems Laboratory a baseline of knowledge when it comes to the implementation of the main logical components of the web application. This section provides insight into the way of development, the functions and methods that are implemented, and what the system can achieve currently. It contains theoretical pointers to the source code, along with aspects of configuration, management, and core parameters within the application.

5.3.1 User

The application deals with each type of user in a specific way, however, there are also some actions that any user can do, which will be explained in more detail in Role section. Package `src\main\java\nl\utwente\ps\inventory\api\user` deals with user-related actions and operations of the Inventory Management System. Both authentication and authorization are embedded in the logic of user-related operations.

Firstly, within the user package lay the data transfer objects for the user logic, namely UserDTO, UserLoginDTO, and UserSignupDTO. The UserDTO contains all the user information that is also stored in the database, except the password field, and is mainly used to provide user information on GET requests. The UserSignUpDTO is used to send all user information to the server when a new account is created. It includes precisely the information that the user is asked to provide when signing up to the Inventory Management System, which is an email, university id, first name, last name, and password. The final DTO, the UserLoginDTO, is similar to the UserSignupDTO, but only contains the fields email and password.

Secondly, the user controller handles all user related requests from the client to the server. These include straightforward logic such as getting all users, getting a specific user by its id, getting all users with a specific role, and deleting users, but also the authentication-related requests are handled in the user controller. These requests include creating an account, log-in and log-out, and changing password. The OTP log-in and change password operations consist of two endpoints. The first requires an email address as input and uses Email component to send an email to the user with further steps to take and a generated code or token to complete the operation. The second endpoint then expects the generated code or token, validates it, and generates the session token or changes the password.

The business layer of the user components handles all the logic related to requests received by the user controller. A unique function in the user service is to get the user id from the session. This method looks at the current session, which is set in Middleware

component to obtain the user who is currently logged in. This is useful as this can be used to know which user is making a certain request or to validate if a certain operation is allowed, without the need for the client to send its user id in those requests.

Finally, the user package contains custom user details and services, this is used in the authentication and authorization process performed in the middleware, and will therefore be explained in Middleware section.

5.3.2 Role

The next component is the Role, which can be found in the module `src\main\java\nl\utwente\ps\inventory\api\role` and has seen some changes in implementation throughout the development.

Initially, Role had its own table in the database with fields for its id, name, and permission level, and it was created with all different layers in place. However, during development, the team noticed that this approach was unnecessary and just too much of a hassle as most of the role request had to be combined with user logic. Therefore, the foreign key field of the role in users would become the entire role component, making it part of the User component (Section 5.3.1). The Role entity became an enumerated, storing the 3 different user role ids and their name, and the role module only contains a RoleDTO and a RoleController.

The RoleDTO is part of the UserDTO and contains an id and name field, which can correspond to the following 3 roles:

0. Guest
1. Employee
2. Admin

The role controller is a simple controller that has two endpoints, one to get all roles and one to get a specific role based on a provided id.

The main purpose of the role component is the different level of permissions in the Inventory Management System, which is explained in 3.2.2. After the authentication process, explained in 5.3.3, the roles are used for the authorization process. Each controller endpoint that requires a specific user role is annotated with `@PreAuthorize("hasAuthority('<Role>')")`, where role can be Guest, Employee, or Admin. This ensures that annotated endpoints are only reachable when users are logged in with an account with that specific role. endpoints marked with Employee can also be accessed by Admins as administrators are also employees in the Inventory Management System.

5.3.3 Middleware

The middleware is not really a component like the other components discussed in this section and could also be classified as a layer, as every request will go through the middleware before reaching any of the controllers. The middleware is located in the module

`src\main\java\nl\utwente\ps\inventory\middleware`.

The first part to analyze is the security configuration, which has two main functions. Firstly, the security configuration contains the (custom) implementation of some of the security related objects, such as the CORS filter which is used to run the client on a different port than the server, the password encoder, and the authentication manager which uses the custom user details and service. Additionally, the security configuration handles the security filter chain, it allows access to certain paths while requiring authentication for others, and it adds two additional custom filters to the filter chain.

The two additional filters are the exception handler filter and the JWT authentication filter, which are both added to Spring Boot's default filter chain. The exception handler is added right before the JWT authentication filter and is solely tasked with handling exceptions thrown by that filter to send a 401 (unauthorized) exception back to the client instead of a unhelpful 500 (internal server error). The JWT authentication filter uses the JWT service to validate the JWT token and custom user details, and the service to authenticate the user on each request.

The JWT service handles all logic regarding the JWT token, such as generating and validating it, extracting the user email, and rotating the secret signature key. The JWT token is generated by providing the user's email address, the expiration date, and the secret key, after which Spring Boot Security will automatically generate the JWT token. The email address and expiration time are part of the token and can therefore be extracted and validated by the secret key. The custom user details service gets the user object from the database and passes it to the custom user details. The custom user details class is used to set the user roles of the current user before storing in the session, so Spring Boot can use it to authorize requests that are annotated with a specific role, as explained in 5.3.2.

More information on how the middleware improves and ensures the security of the Inventory Management System can be found in 7.

5.3.4 Supervisor

An important concept in the Inventory Management System is the supervisor. Since guests cannot directly borrow items themselves, they need an employee or administrator as their supervisor to accept and allow their borrowing transactions. This is managed in the package `src\main\java\nl\utwente\ps\inventory\api\supervisor`, which contains the data transfer object, the controller, and the service.

Firstly, the supervisor DTO consists of the following fields: guest id, employee id, status, request date, and response date. The guest id and the employee id together create the id of the supervisor entity. The status indicates whether a supervisor request is accepted by the employee and can be one of these 3 states: Pending, Approved, or Rejected. The request date and the response date are stored as longs and indicate when a supervisor request is created and when it is accepted or rejected.

Secondly, the supervisor controller has endpoints for getting, creating, and deleting super-

visor relations between users. These endpoints are split into guest endpoints and employee (or administrator) endpoints, for example, guest endpoints start with `/api/supervisor/user` and employee endpoints start with `/api/supervisor/employee`. Employees have an additional endpoint they can use, which is accepting or denying a specific request.

Lastly, the supervisor service handles all the business logic behind the endpoints in the controller. The service makes clear why the controller is divided into user and employee parts, while still having the same functionalities and endpoints. The first reason is that in this way in the service the same method can be used, while still making sure that each user is correctly identified as the guest or employee. Most endpoints obtain one of the users, the one that makes the request, via the session as explained in section 5.3.1, and the other user is provided in the body of the request. Depending on the order in which they are provided as parameters to the supervisor service methods, it knows which user is the guest and which is the employee. The second reason is that some operations are slightly different depending on if a guest or an employee performs them. For example, when a guest creates a supervisor request, it status is set to Pending, while as when the employee creates it, it does not have to be accepted by the employee again, and therefore the status is automatically set to Approved.

5.3.5 Category

This is one of the crucial components of modeling hardware items within the Inventory Management System. The categories allow certain types of variable to be stored for multiple collections of items, establishing the scalability of the system. They are managed in the module `src\main\java\nl\utwente\ps\inventory\api\category`. This module contains the data transfer object for categories and general variables.

Firstly, the category DTO is comprised of a base class which includes only the name and which allows for retrieving the category names without unnecessary information. Then, there are additional DTO aimed at fitting specific needs when it comes to data migration to the user interface. CategoryDTO is an extension of BaseCategoryDTO that encapsulates the list of variables within the category.

Secondly, the category controller supports getting all category names, creating a new category, retrieving it by id, updating it given an updated category DTO, and deleting it by id. Moreover, the category controller handles the CRUD operation of general variables. General variables are ones with only the known name and type. The variable service is also part of the category module because these two components of the system are intertwined, together forming a specifications system of any hardware item. Categories can have sets of variables of a variety of types.

5.3.6 Type Variable

The type variable is an abstract data transfer object that encapsulates a variable of any type with its value, mapped to a general variable and a collection. This means that the type variable consists of the value of a certain variable in the context of a collection. For instance, let there be a category "Battery" that has variable "Voltage". Let there also be a collection of batteries, which would have 5V value for voltage. Thus, the variable

including 5V is a "TypeVar" as it maps the general variable Voltage which exists within the battery category, with the collection of batteries, part of same category, encapsulating the value of voltage specific for the collection.

Type variables are determined by the identifier of a variable along with the identifier of the collection associated with the value inside. An important aspect is that, in order for the type variable to exist, the category of the collection associated with it needs to have a variable of the same type as the value given.

There are four types of TypeVar objects, each representing the possible types of technical specifications one can find in a hardware. The logic of the type variables can be found in the module `src\main\java\nl\utwente\ps\inventory\api\typevar` and contains data transfer objects for each type of variable,namely BoolVar, IntVar, DoubleVar, StringVar. Each of the classes has a custom mapping from an integer id to the type. In that sense, '0' is mapped to BoolVar, '1' to IntVar, '2' to DoubleVar and '3' to StringVar. This mapping can be found in TypeVarTypeIdResolver.

5.3.7 Project

The business logic regarding the projects can be found in the package `src\main\java\nl\utwente\ps\inventory\api\projects`. The projects within the system aim to model the Pervasive Systems Research projects, and they are often linked to a contact user.

To model this, data transfer objects are classified into three categories, namely base, create, and complete project. The BaseProjectDTO abstract class contains the common fields found in all other data transfer objects. Then, the application has one data transfer object with the user object in its entirety, to send to the user interface as a response when retrieving the project. The last type of data transfer object is CreateProjectDTO, which abstracts unnecessary fields of the contact user and requires only its identifier. This class is important to not require the client to first fetch the entire user by id and then request creation of a project. This operation is dealt with in the business logic of the web application, thus maintaining the separation of layers and concerns.

The project service and controller support the creation, retrieval, updating, and deletion of projects. When creating a project, one needs to link it to an existing user; therefore, the existence of the contact person is a prerequisite. Moreover, the service supports retrieval of all projects, of a project by its identifier, and also allows checking whether a project exists by the identifier. Lastly, the controller contains a project deletion method. For both the creation and deletion of a project, the user issuing the request must be an employee of Pervasive Systems. Lastly, the service allows for updating a project; however, it was deemed unnecessary to include it into the application interface; therefore, the controller currently does not use the updating method.

5.3.8 Item Collection

An important aspect when discussing the implementation of the Inventory Management System is how the system handles hardware items. Their management happens within the package `src\main\java\nl\utwente\ps\inventory\api\collection`. This pack-

age contains both the individual item and the collection item management.

Firstly, it is necessary to repeat the items that are supported by the system. The collections of items can be unique, bulk, and infinite. A collection of unique items models an item collection for which the condition and state of each item shall be monitored, and each hardware item has its own unique identifier. All these unique items are managed by the ItemService, ItemController and represented via ItemDTO. The ItemDTO contains the information needed for each item, namely the identifier, collection of which the item is part, the condition, and the status. The condition of an item can have values: good, damaged, broken; whereas the status can be: available, borrowed, or unavailable. These facilitate the state monitoring of hardware items, a key feature of the Inventory Management System. The service and controller for unique items support adding, updating, retrieving, and deleting of unique items.

Secondly, hardware items are primarily managed in collections. The collection controller and the service deal with the different types of collections in separate ways. It is important to note that, similarly to the project, the collection data transfer objects consist of a BaseCollectionDTO which contains the common fields among the collection transfer objects. This abstract class is extended, then, by CollectionDTO which is mainly used for return responses and management of collections, and contains additionally the category and project objects. Another extension of the BaseCollectionDTO is the CreateCollectionDTO which is used in collection creation, abstracting the need for the upper layers to provide the entire category and project objects when issuing the request, the identifier of both being enough. Moreover, the CreateCollectionDTO contains an additional field, namely the list of all values of variables for this collection.

When creating an item collection, it is necessary that the name is not empty, and that the available and total quantities are positive integers. The collection service allows the creation of hardware item collections by first checking the variables given by upper layers if they are associated with the category of the created collection. Secondly, the method retains the total quantity given by requester and then deals with creating the collection according to its type. At this stage of the creation process, if the collection is identified as unique, the method will generate items corresponding to the specified total quantity, thus ensuring that the repository of items is updated. However, if the collection is bulk, then the system will retain the total quantity and will initialize the available quantity of items with the same value. It is an assumption within the system that when creating the collection, the person will set the total quantity as the number of currently available items, and consequently the IMS will keep track of their availability. The creation method needs to use both the ItemService and TypeVarService to ensure that adding of items and variables is consistent throughout the system and that they are linked to the created collection.

Both the service and controller support the retrieval of item collections by id, by name, and fetching of all collections. Moreover, they allow items of a collection to be returned. Another important operation supported is adding items to a collection, either bulk where a count is incremented or unique, where more items are created. Similarly, variables, along with their values, can be queried and updated. Lastly, the collection counts can be updated in case the administrator orders more pieces of certain items within a collection.

5.3.9 Filtering

One of the more interesting features of the Inventory Management System is the way it deals with technical specifications of hardware collections. As specified before, the item collection contains a number of items with the same names and specifications; therefore, a good way to help the staff of Pervasive Systems in managing their hardware is to filter through the item collections. The collection filtering is part of package `src\main\java\nl\utwente\ps\inventory\api\filtering`.

In this paragraph, the data transfer objects will be discussed. The first is CollectionSearchCriteriaDTO, which consists of nullable fields that may be used in searching for a collection. For instance, one may search a collection by its vendor, or name and project name. It is the choice of the upper layers how many of these criteria are used in searching. Moreover, there is need for abstraction such that variables contain only the name and value, to facilitate the filtering, thus the data transfer object for variable is redefined by FilterVarDTO.

The filtering service and controller facilitate the retrieval of collections based on advanced search criteria such as categories, values of variables, and fields. As previously specified, the search-based criteria is designed to be flexible, by allowing any combination of fields to be set. Fields with null values are disregarded by the method, thus it will return collections that match the values of the fields that were explicitly set. Another feature implemented, is the filtering of collections based on certain values of variables. This would allow users to search for Arduinos featuring a specific processor, or with a certain amount of GPIO pins. The method receives a list of FilterVars and it retrieves the collections with same values for variables as in the list. Lastly, filtering by category is straightforward, it will return the collections part of a specific category by the category identifier.

5.3.10 Borrow

The BorrowItem component represents individual item-level records within a borrowing request. While a BorrowRequest manages the overall workflow of a borrowing transaction, BorrowItem ensures item-level tracking for inventory units that require individual accountability. This component plays a crucial role in ensuring traceability, managing returns, and preserving a complete history of item usage.

They are managed in the module `src\main\java\nl\utwente\ps\inventory\api\Borrow`. This module contains the data transfer objects for individual borrowed entries, the lifecycle management service logic (borrowing, returning, and reporting damage), and the database logic to assign items to users.

Firstly, a BorrowItem is created only for inventory items marked as unique. These items represent physical trackable units that must be accounted for individually (e.g., specific laptops, sensors, or devices with unique serial numbers). The BorrowItem entity captures key details, including the linked hardware item, borrow start and end dates, return timestamp, and an optional flag indicating whether the item was returned in a damaged state.

Secondly, the borrowing process is designed to reflect the role-based access control of

the system. Regular users, such as students and supervised employees, must initiate a BorrowRequest that includes one or more BorrowItems for unique items. These requests are subject to approval of the supervisor. Upon approval, the associated borrow items are activated, and the borrowing is officially tracked per item.

In contrast, users with elevated permissions, namely administrators and employees, have the ability to borrow items directly without going through the request-approval workflow. This direct borrowing path is intended for operational flexibility and urgent administrative use.

However, for items that are not marked as unique, such as those classified as bulk (limited quantity) or infinity (unlimited supply), a BorrowItem record is not created. Instead, when such items are borrowed, the system updates the available quantity value in the corresponding collection entry. This approach reduces unnecessary record creation for consumables or frequently reused stock while still maintaining accurate availability status at the collection level.

The borrow item controller provides endpoints for retrieving borrowed items, processing returns, and submitting damage reports. It supports partial returns for multi-item requests, and it allows the system to update return states independently for each item.

The service layer ensures correct lifecycle handling for each item, with clear logic depending on the item's type and the user's role. This hybrid approach, using explicit item records for unique assets and quantity tracking for bulk items, offers both flexibility and accountability, aligning the implementation with the practical management needs of a research lab environment

5.3.11 Damage

The damage reporting is implemented in the package: `src\main\java\nl\utwente\ps\inventory\api\damage`. The damage report data accessing objects are modeled similarly to projects, namely there is a base abstract class that contains the common fields, and two extensions of this class, one containing the instance of an object and the other abstracting it to only contain the identifier of the object. In the case of damage reporting, each damage report is based on a borrowing of an item. It is a prerequisite for a user to borrow an item in order to report it as damaged. Therefore, the ItemDamageDTO will contain a BorrowItem object, whereas CreateItemDamageDTO will only have the borrow identifier.

Furthermore, the service and controller of damage reports support adding,updating reports along with changing the condition of an item. It will update the state of the item within the unique item repository. The damage condition fields are defined as an enumerated type, which can assume one of the following values: minor, moderate, severe, or irreparable. However, the item condition within the item objects can take the following values: good, damaged, broken. Therefore, when reporting a damage condition as 'minor' or 'moderate', it means that the item is still usable and will have the inner condition 'damaged'. Similarly, if an item is reported with severe or irreparable damage, the item is flagged as broken within the item repository.

Lastly, the service includes functionality for retrieving all damage reports by an item, to show the user the past reports of an item. It also allows retrieval of all or one damage report by identifier.

5.3.12 Email

The implementation of email can be found in the package `src\main\java\nl\utwente\ps\inventory\api\email`. Firstly, the model for email consists of an abstract class BaseEmailDTO. It contains identifier of the user and subject. Its extensions are the EmailTemplateDTO and the EmailDTO. The former is a default template containing variables such as assets, subject, user, address, whereas the latter contains only the message. The controller allows only for sending emails, with the administrator permissions.

5.3.13 Utils

Util classes are helpers aimed at performing background or side tasks, tasks that do not directly concern the current scope when needed. One such task is the declaration of a JPA specification for filtering. The class contains custom queries and helps with filtering collections by fields. The util classes can be found in the package `src\main\java\nl\utwente\ps\inventory\util`.

Chapter 6

Testing

6.1 Usability Testing

6.1.1 Methodology and procedure

To evaluate the usability and efficiency of the Inventory Management System (IMS), the development team conducted task-based user testing with stakeholders closely related to the Pervasive Systems Research Group. Participants included PhD candidates and employees from the group, as well as our own development team of six students who acted as borrowers in the student role. Due to time constraints, it was unfeasible to include guest students into testing sessions. We designed realistic tasks based on the user and functional requirements of the system. Each task was aligned with an actual use case and described in a goal-oriented way, avoiding any UI-specific terminology. Participants were asked to complete tasks using the system without guidance to better assess intuitiveness and overall usability.

To reduce potential bias and ensure consistency, we followed four core principles during all user testing sessions:

- **Neutrality:** Tasks were stated as general objectives, without revealing interface elements or suggesting a specific action flow.
- **Text Communication:** All task instructions were provided in written form. If clarification was required, it was recorded as an intervention.
- **Observation Only:** Researchers observed the task execution without interference. Notes were taken after task completion.
- **Documented Intervention:** If participants encountered difficulty and required help, this was clearly recorded for later analysis.

Qualitative data was collected through direct observation and semi-structured interviews after the testing sessions. In some cases, participants used the "thinking-out-loud technique to verbalize their thought process. The resulting feedback and behavioral patterns were analyzed using thematic analysis and affinity diagramming to identify usability issues and improvement opportunities.

6.1.2 Task-based Testing

The development team has separated the tasks for different stakeholders. First, they will presented the tasks for borrowers, including different students and employees. Then, the tasks specific for supervisors and administrators. The tasks have been selected such that they reflect the main functionality of the application.

Tasks for Borrowers (Students and Employees)

In this section, we present the tasks assigned to student and employee users. These tasks were designed to evaluate key features for borrowing.

Task 1: Borrow an item
"You need a Raspberry Pi for your sensor experiment. Please borrow it from the system."
Participants were generally able to find the correct item and complete the borrowing process in 6-8 clicks. Some confusion was reported around setting the return date, which suggests that clearer UI cues or default options would be helpful.

Task 2: View your borrowed items
"You want to check what items you currently have borrowed."
All participants successfully accessed their dashboard or "Borrowed Items" section with minimal effort. Navigation was intuitive.

Task 3: Report a damaged item
"You realize one of your borrowed sensors is damaged. Please report the damage in the system."
Users found the damage report functionality, though a few suggested that the option should be accessible directly from the "Borrowed Items - Return report" page.

Tasks for Supervisors

In this section, we provide tasks that simulate the responsibilities of supervisors who manage student borrowers. These tasks test features such as reviewing requests and monitoring borrowing activity.

Task 1: Approve a borrowing request
"You received a borrowing request from a student. Please review and approve it."
Most supervisors completed this task easily. One participant initially looked in the wrong section, indicating that the supervisor dashboard might benefit from clearer labeling.

Task 2: View borrowing history of your students
"Check what items your supervised students are currently borrowing."
All participants were able to perform this task. A minor suggestion was made to include a search-by-name feature.

Tasks for Administrators

These tasks evaluate administrator-specific functionality such as managing item categories, creating collections, and editing inventory content. The goal was to assess how intuitive and efficient the backend management tools are.

Task 1: Create a category
"Create a new category called 'Power Supplies'."
All admins completed this task without difficulty. The form was clear, though suggestions were made to include default examples.

Task 2: Create a collection and add variables
"Under 'Microcontrollers', create a collection called 'ESP32 Family' with variables: voltage, Wi-Fi support, cores."
This task was performed successfully. Most participants found the workflow logical and appreciated the flexibility.

Task 3: Add a new item to a collection
"Add a new ESP32-WROOM board to the ESP32 Family collection."
Task was completed smoothly. Participants found the process structured, though a few requested copy functionality for similar items.

Task 4: Remove item, collection, and category
"Delete the ESP32-WROOM item, then remove the 'ESP32 Family' collection and the unused 'Power Supplies category."
Participants were able to remove these items, though some mentioned the delete options were somewhat hidden.

6.1.3 Unit Testing

Unit testing was conducted throughout the development process to ensure that individual components of the Inventory Management System (IMS) performed their intended functions correctly and independently. These tests were focused on core service logic, helper utilities, middleware, and controller behavior under isolated conditions.

6.1.4 Tools and Frameworks

All unit tests were written in Java using the JUnit 5 framework. Mockito was used for mocking dependencies, particularly in service-layer testing, to isolate business logic from external systems like the database. The tests were run and analyzed using IntelliJ IDEA's built-in coverage tooling.

6.1.5 Coverage Overview

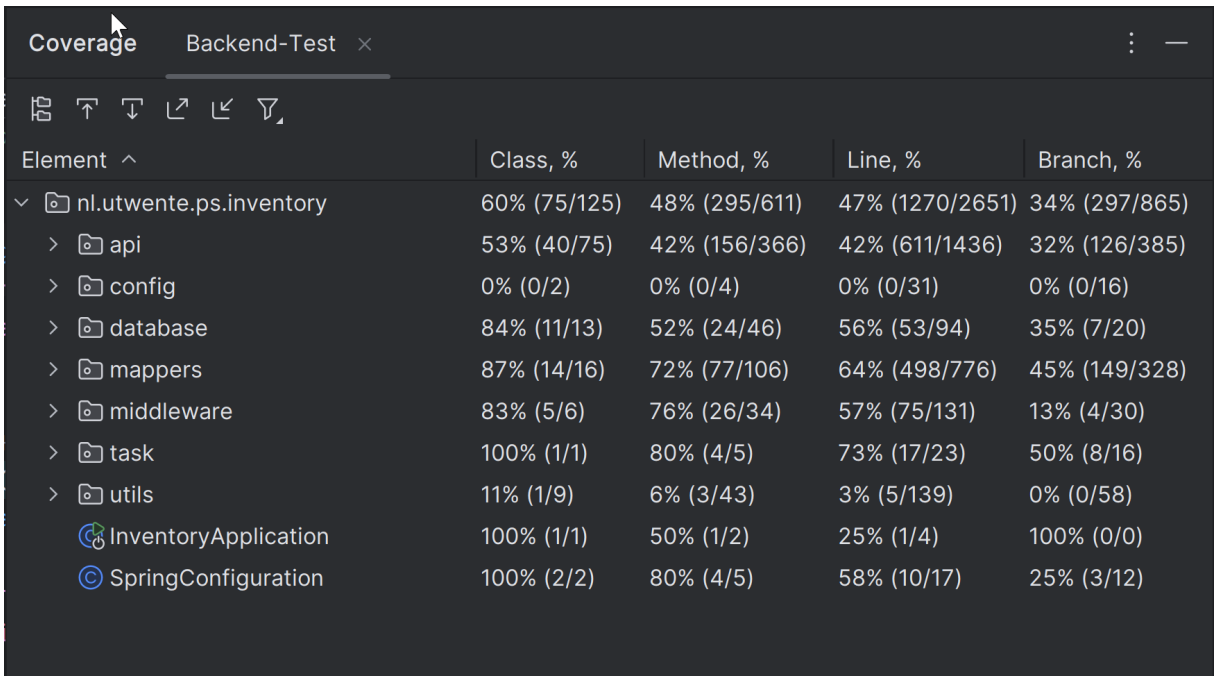


Figure 6.1: Testing Coverage Overview

The codebase achieved a partial but meaningful level of unit test coverage:

- Overall class coverage: **60%** (75 out of 125 classes)
- Method coverage: **48%** (295 out of 611 methods)
- Line coverage: **47%** (1270 out of 2651 lines)
- Branch coverage: **34%** (297 out of 865 branches)

Detailed Coverage by Module

- **api**: 53% class, 42% method, 42% line coverage. Focused on controller responses and request mappings.
- **config**: 0% coverage. Excluded from testing due to static configuration annotations.
- **database**: 84% class, 52% method, 56% line coverage. Tested repository interfaces and entity logic.
- **mappers**: 87% class, 72% method, 64% line coverage. Mapping between entities and DTOs was thoroughly verified.
- **middleware**: 83% class, 76% method, 57% line coverage. Tested filters, interceptors, and exception handling.
- **task**: 100% class, 80% method, 73% line coverage. Covered scheduled background operations.

- **utils**: 11% class, 6% method, 3% line coverage. Limited testing of utility functions.

The `InventoryApplication` and `SpringConfiguration` classes achieved full class coverage and moderate method/line coverage, ensuring proper context initialization.

Testing Approach

Each test case followed the Arrange-Act-Assert pattern:

1. **Arrange:** Mocks and test data were created using Mockito and builder patterns.
2. **Act:** The target method was executed with defined inputs.
3. **Assert:** Outputs and side effects were validated using assertions or mock verifications.

Service-layer tests focused on verifying business logic, including:

- Borrow request lifecycle (creation, approval)
- Inventory filtering and search functionalities
- Role-based access control enforcement
- Data collection and variable assignment logic

6.1.6 Limitations and Future Improvements

While coverage is strong in core service and middleware layers, some packages such as `utils` and `config` received minimal attention due to their structural nature or lower risk. Improvement efforts will include:

- Adding MockMvc tests to boost `api` layer coverage.
- Increasing coverage for utility methods and edge cases.
- Exploring mutation testing for robustness validation.

6.1.7 Conclusion

The unit tests lay a reliable foundation for validating the system's core functionality. Although full coverage is yet to be achieved, the current test suite effectively safeguards critical components, and future enhancements will continue strengthening test quality and maintainability.

6.2 Integration Testing

Integration testing was performed to verify the proper interaction between different components of the Inventory Management System (IMS), particularly between the front-end and back-end via REST APIs. This level of testing ensured that user actions in the interface correctly triggered the intended backend operations and that responses were rendered properly in the UI.

The main goal of the integration tests was to validate whether the system components work together as expected, including data flow, role-based views, and the management of user actions such as borrowing, approving, and updating hardware items.

6.2.1 Methodology

Integration testing was carried out manually by simulating typical user scenarios and observing how the interface interacted with the API and the database. A pre-filled test database was used to replicate realistic situations. We followed a task-based testing approach to evaluate end-to-end workflows. Each test scenario was executed on a deployed development version of the system.

Tested Workflows

Several end-to-end workflows were selected for integration testing:

- Borrower searches for an item, submits a borrow request, and checks their borrowed items page.
- Supervisor reviews a borrowing request, approves it, and verifies the updated status.
- Administrator creates a category, collection, and item, then removes them in reverse order.
- Borrower reports a damaged item and confirms it appears in the system for the supervisor or admin to review.
- User logs in, performs an action, logs out, and logs back in to verify state persistence.

Each scenario was tested to ensure that:

- Front-end requests were sent correctly with appropriate HTTP methods (GET, POST, PUT, DELETE).
- Backend services correctly handled business logic, including authorization and validation.
- The database state was updated consistently.
- The front-end updated dynamically and reflected server-side changes accurately.

Findings and Observations

Integration testing revealed that the core workflows functioned correctly, with consistent data synchronization between the user interface, backend services, and the database.

A few minor issues were observed:

- After item creation, the new item list did not refresh automatically, requiring a manual reload.
- Error messages were not always displayed clearly when invalid form data was submitted.
- In some cases, deleted categories or collections remained cached in the interface, leading to temporary display inconsistencies.

These issues were logged and addressed during further development, and improvements were made to state management and error handling in the front-end.

Conclusion

The integration testing phase validated that the various components of the IMS communicate effectively and that the system behaves as expected in realistic scenarios. With minor improvements to user feedback and interface reactivity, the system was considered robust and well-integrated, ready for further testing and deployment.

6.3 API Testing

API testing was conducted to ensure the correctness, reliability, and security of the RESTful endpoints that connect the front-end with the back-end services in the Inventory Management System (IMS). These tests verified that the APIs return appropriate responses under various conditions, including both expected and edge cases.

Objectives

The main objectives of API testing were:

- To verify that all endpoints return correct HTTP status codes and response bodies.
- To ensure that user roles (guest, employee, supervisor, administrator) receive appropriate access and permissions.
- To validate input data through server-side constraints.
- To check the consistency and accuracy of CRUD operations performed on inventory entities such as users, items, categories, collections, variables, and borrow requests.
- To assess error handling and response messages for invalid inputs and unauthorized access.

Testing Tools and Environment

Postman was used for manual testing of REST APIs. Automated tests were written using Spring Boot's built-in testing tools, such as `@WebMvcTest`, `@SpringBootTest`, and `MockMvc` for controller-level verification. The test environment included:

- A clean test database initialized before each test suite.
- Authentication via JWT for protected routes.
- Sample user accounts for each role (guest, employee, supervisor, admin).

6.3.1 Tested Endpoints

The following endpoint categories were tested:

- **Authentication:** Login, logout, and session handling.
- **Users:** Role-based user access, viewing and managing user profiles.
- **Items:** Adding, editing, deleting, and retrieving items assigned to specific collections.
- **Categories:** Creating, updating, and deleting hardware categories.
- **Collections:** Managing logical groupings of items within categories.
- **Variables:** Creating and associating technical attributes (e.g., voltage, memory, cores) with item collections.
- **Borrowing:** Creating borrow requests, returning items, checking status, and handling overdue items.
- **Reporting:** Submitting and viewing damage reports for borrowed hardware.

Results and Observations

API testing confirmed that all tested endpoints behaved as expected across all roles. Input validation was consistently enforced, and access controls were effective in preventing unauthorized actions. The hierarchical relationship between categories, collections, variables, and items was well-maintained in both API structure and database behavior.

Some improvements were made during testing:

- Improved error messages for variable type mismatches and missing associations during item creation.
- Added cascade deletion checks to prevent accidental data loss when removing categories or collections.
- Refined permissions to restrict variable management to admin users only.

Conclusion

The API layer of the IMS is stable, secure, and REST-compliant. CRUD operations for all major components—categories, collections, variables, items, and users—functioned correctly. Validation and authorization mechanisms ensured proper data integrity and user restrictions. The system is well-prepared for integration with the production front-end and supports scalable management of complex hardware inventories.

6.4 Security Testing

Security testing was performed to evaluate the robustness of the Inventory Management System (IMS) against common vulnerabilities and unauthorized access. Given that the system handles sensitive data such as borrowing records, personal information, and administrative privileges, it was crucial to ensure that all security measures were correctly implemented and enforced across the application layers.

Objectives

The primary objectives of security testing were:

- To validate that all role-based access controls (RBAC) were correctly enforced.
- To prevent unauthorized access to protected endpoints.
- To ensure secure user authentication and session management.
- To verify input validation and prevent common attacks such as SQL injection and Cross-Site Scripting (XSS).
- To assess secure data handling and error message sanitization.

6.4.1 Tested Areas and Methods

The following security areas were tested:

- **Authentication and Authorization:** JWT-based authentication was tested to ensure that access tokens are securely generated, stored, and validated. Role-based access to REST endpoints was verified by attempting actions with insufficient permissions and confirming proper HTTP 403 Forbidden responses.
- **Session Handling:** Expired and malformed tokens were tested to ensure proper rejection and redirection to login. Token tampering and replay attacks were attempted using Postman to simulate invalid sessions.
- **Access Control:** Users were prevented from accessing or modifying data outside their roles. For example, borrowers could not access administrative endpoints, and supervisors could not delete items or collections.
- **Input Validation:** All form inputs, especially in category, collection, variable, and item creation endpoints, were tested against injection attempts. Data was validated server-side, and special characters were sanitized.

- **Error Handling and Exposure:** API responses were reviewed to ensure no internal stack traces or sensitive information (e.g., database structure) was exposed in error messages.
- **HTTPS Enforcement:** The development server was tested with HTTPS configurations to simulate secure data transmission. Though HTTPS is expected to be fully enforced in the production deployment, local configurations were verified.

6.4.2 Findings and Improvements

The system's security mechanisms functioned correctly under normal and adverse conditions. Specific findings included:

- All unauthorized access attempts were correctly blocked with appropriate HTTP responses.
- JWT tokens were validated for expiry and integrity. Tampered tokens were reliably rejected.
- Role-based restrictions were strictly enforced at both controller and service layers.
- No SQL injection vulnerabilities were found due to the use of parameterized queries via Spring Data JPA.
- Custom error handling prevented stack traces from leaking through API responses.

A few minor security refinements were made:

- Improved feedback for failed login attempts to avoid leaking information about valid usernames.
- Added more specific exception handling for certain access-denied cases.
- Updated CORS policy to restrict access in non-production environments.

6.4.3 Conclusion

Security testing confirmed that the IMS meets best practices in modern web application security. Role-based access control, input validation, secure token handling, and user authentication were all verified. The system is well-protected against common web threats and is prepared for secure deployment within the University's infrastructure.

Chapter 7

Security

7.1 Risk Assessment

This chapter focuses on the security risks associated with this system, highlighting the dangers involved in a web application and database. It will describe the vulnerability each of these threats poses. The following chapters will describe how risks will be prevented and what security measures are implemented to minimize vulnerabilities.

SQL injection

An attacker can enter malicious SQL code in the input fields, which could lead to the system executing unauthorized commands. This can for example be used to gain unauthorized access or manipulation of data in the database.

XSS attacks

In an XSS attack, an attacker tries to inject malicious scripts into the system. These scripts could be executed in other user's browsers, possible resulting in sessions being taken over or sensitive information being stolen.

Weak or improper authentication

When login logic is not solid or weak passwords are used, an attacker can easily gain access to other user accounts, resulting in unauthorized access to system functions.

Sensitive data exposure

Another risk for the inventory management system is the exposure of sensitive data. If confidential data is not properly protected it allows for attackers to easily gain access to unauthorized information.

DoS and DDos attacks

In a Dos or DDos attack, an attacker aims to overwhelm a website or server with traffic or request. This can lead to the application being unavailable for users, resulting in dissatisfied users, possibly financial losses and reputation damage.

Insecure API

When the API is implemented insecurely, attackers can possibly bypass the usual security by exploiting the vulnerabilities of the insecure API, resulting in unauthorized actions that can be performed or sensitive data that can get exposed.

7.2 Security by Design

Security by design refers to the approach of designing an application with security in mind, ensuring that the product stays secure throughout its production cycle. Security by design consists of several principles that are important for software development (Ebad, 2022). This section will highlight some of the security by design principles that have been used in the development of the inventory management system.

Secure frameworks and libraries

One of the first step in designing the inventory management system in a securely matter has been deciding what frameworks and libraries to use, and the development team has settled on a set of frameworks that provide the necessary security options to achieve the desired level of security in this system. Section 3.2.3 discusses the choices in more detail, and in the next section it will be clear how these choices influence the security of the inventory management system.

Complete mediation

The principle of complete mediation says that every single request should be properly authenticated and authorized. To achieve this goal, the development team made sure to implement a middleware layer, which is responsible for satisfying the goal of authenticating and authorizing every request. Section 5.3.3 explains the middleware in more detail, and in sections 7.3.2 and 7.3.3 the authentication and authorization security measures are further discussed.

Fail-Safe defaults

An important security by design principle that is implemented in this project is fail-safe defaults, this principle states that every action will result in no access or denied, unless access is explicitly provided. This does not only mean that users should be authenticated and authorized at any time, but also that when the system fails to do so or any other unexpected failure occurs, the system by default denies access rather than permitting it. The development team made sure that if an exception or failure occurs that the current procedure will immediately be stopped and an error response is send back to the user interface. Furthermore, when a failure occurs while accessing the database, any changes that were made during that transaction are reverted and the database is restored to its previous state.

Separation of concerns

Another crucial security by design principle integrated in the development of the system is the separation of duties. Each component in the system belongs to a specific layer, as explained in section 5.2 and visualized in figure 7.1. This ensures that each component has a single responsibility, they handle a specific part of their layer's tasks, resulting in a safer and more secure system.

Security testing

Throughout the development process, the development team has made sure to test the system on its security. Section 6.4 further discusses the security tests and explains how the security tests were done. In the beginning the security tests gave insight in what security measures should be implemented to make the system secure. Later during development, when the security features were mostly implemented, the security tests exposed the flaws in the security features and indicated what parts needed more work.

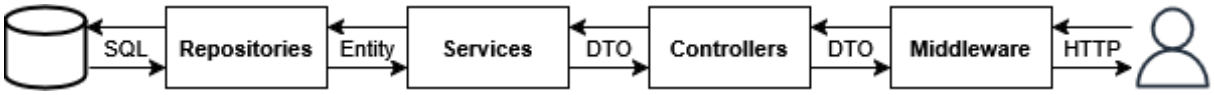


Figure 7.1: Abstract overview of the system layers

7.3 Security Model

In this section the security model will be described, this will explain the measures that are implemented to prevent the threats from impacting the security of the system. This section will be split into different aspects of the security model and highlight how each of them handles a specific aspect of the security.

7.3.1 Threat model

As mentioned in section 5.1, a web application connected to a database introduces some possible exploits. This section explains the threat model that handles the following risks: SQL injection and XSS attack.

Sanitize inputs

Input sanitization refers to the process of modifying or removing potentially unsafe user input. An example of input sanitization would be escaping special characters so a malicious user cannot add a script tag into an input field and make it run on the system. To handle such a threat, the system uses transfer objects to send data between the client and the server. In these data transfer objects, the attributes are defined with requirements, such as not being null or being valid, to ensure the safety of user input of any request.

Prepared statements

Prepared statements refer to reusable SQL queries, which is parameterized and ensures that the query is pre-compiled and only later the user input is added to the final query. Using prepared statements ensures that the user cannot input logic in the query that alters its functionality, but instead only is able to add the expected values. In the system this is handled by the repositories, these repositories use prepared statements to get or alter data from and in the database.

XSS protection

On top of the previously mentioned security measures, protection against XSS attacks was one of the few specifically security related measures that can be found explicitly in the requirements, as can be read in chapter 2. Therefore, this vulnerability receives some special attention, specifically in combination with the chosen frameworks. React, the library used for the client application interface, automatically escapes special characters to harmless HTML entities, to protect against possible XSS attacks (Gamma & Gerasimenko, 2024). As highlighted by Gamma and Gerasimenko (2024), React is generally called XSS protected and with the some careful consideration a web application will be protected against XSS attacks. Additionally, within the security configuration of spring boot, the framework used for the backend, the development team has enabled XSS protection. With this enabled, it will automatically block any request that look suspicious.

7.3.2 Authentication

A common, and generally required, security measure for a web application is user authentication. By authenticating users it is possible to provide them with the information that is relevant for them, and it will keep personal information private from other users. In this section, the different methods of login will be discussed, as well as the authentication token that is used to authenticate user's requests to the server.

Username and password login

The first method of login in the inventory management system is the well known username and password authentication. When creating the account, a user is asked to input, among others, an email address and a password. When accessing the system, users are simply asked to enter their email and password, which will be validated by the server and in return provide an authentication token.

One time password login

The second method of login that is supported is a one time password. A disadvantage of the username and password based login system is that people tend to forget these or create really simply and insecure ones out of fear to forget them (Adams & Sasse, 1999). Similarly, the stakeholders did not expect to use this system very often and therefore did not want to remember a password, therefore allowing to login using a one time password became one of the requirements (Chapter 2). To login using a one time password, a user first enters their email address, then the server will generate a random 4 digit password, which is send to the provided email address and is valid for 5 minutes. This 4 digits can then be entered on the website as well, and after validation the authentication token will be provided again.

JWT authentication

When using the system, users will send requests to the server when they try to access, manipulate or delete data from the database. To authenticate the user all those requests will send their JWT token, which is the authentication token that is generated when a user uses one of the login methods. The generation process of the token is explained in chapter 5.3.3. To validate user authentication, the JWT token gets parsed to be able to

extract the user email address. During the extraction process, the token signature (the randomly generated secret key which has been used to sign the token) and expiration time get validated. Once the token is validated and the email address is extracted, the user is authenticated. To make this authentication process more secure, the tokens have an expiration time of 3 hours, after which a user will have to login again. Additionally, the secret key used for the token signature are rotated every day, increasing the security aspect.

7.3.3 Authorization

As already explained in previous chapters, in the inventory management system not all users will have the same permissions. This section will highlight the system's role-based access control, its impact on the security of the system, and how this is handled.

Role-based access control

Users of the inventory management system will be authorized based on their role in the system. Depending on whether a user has an administrator, employee, or guest account, they will or won't be able to make certain request. As explained in detail in section 3.2.2, guests are primarily able to view pages, while employees and admins are also able to add new items, projects, or categories to the system.

Permission based request

The role-based access control allows for permission based requests. Every request that is send to the server will reach its desired endpoint after passing through the previously discussed security measures. Each endpoint that requires a specific permission level of the user is marked with an annotation that specifies which role is required. During the authentication process described in the previous section, the role of the user will be set for the current section, allowing Spring Boot to compare the user's role with the required permission level. When the user does not meet the permission requirement, the endpoint will not be triggered, instead a 403 (forbidden) response will be returned to the client, which will inform the user that this action is not allowed, resulting in a secure and complete authorization process.

7.3.4 Data protection

Data protection is a crucial part of the system's security, especially because it stores personal and confidential information about the users. This section will highlight some measure that were taken to protect user data.

Password hashing

One specific instance of data protection is hashing passwords. User passwords should be stored securely, therefore, they get hashed before being stored in the database. This is done by Spring Boot's delegating password encoder, which can use different encoding algorithms for different passwords, with Bcrypt as the default algorithm.

Restricted database access

To ensure data protection in the inventory management system, the access to the database is restricted to the server. More specifically, the repository layer is the only layer in the system that directly communicates with the database, as described in section 5.2 and visualized in figure 7.1. Therefore, this is the only layer that retrieves, alters, and delete data in the database. This keeps data protected, as before interacting with the database all other measures previously described in this chapter have to be faced, ensuring maximum security.

Chapter 8

Discussion and Process

8.1 Planning

8.1.1 Implementation Trajectory

Weeks 1-3

During these initial weeks, the project group will conduct meetings with stakeholders to finalize and validate the system requirements. This will ensure that the approach of the given problem is aligned with the needs and expectations of both the client and the end users.

The Design phase will include creating mock-ups and wireframes in Figma demonstrating the look and feel of the user interface, ensuring usability and clarity. The design will be presented to stakeholders for feedback.

At the same time, UML diagrams will be created to visualize the architecture and structure of the system, covering both the front-end and the back-end.

This is the first round of user confrontation regarding the requirements and prototype. Usability tests will be performed in this phase to assess whether the interface is satisfactory.

Weeks 4-5

These weeks the development team's main focus is to implement the logic and functionality of the web application. The back end will be implemented using a scalable and secure framework supports the system requirements. This will include setting up the server-side components and API endpoints.

The database development process will include the design of a schema to facilitate the structured storage of inventory data, user information, borrowing transactions, and other relevant records. A relational database, e.g. MySQL or PostgreSQL, will be used to ensure consistency and reliability.

The integration of user roles and authentication mechanisms (via the Authenticator app) will be considered to ensure the system's privacy, authentication, and integrity.

Weeks 5-7

Over these weeks, the front-end implementation and development will focus on designing a user-friendly and responsive interface using the React.js and Bootstrap frameworks.

React component-based structure will help organize the user interface efficiently, while Bootstrap will supply pre-designed components and a responsive grid system to ensure compatibility across multiple platforms.

When designing the system, the development team aims to make components intuitive with features such as item availability tracking, the borrow/return process, and notifications for overdue items. Particular emphasis will be placed on the search and filtering features of the inventory, ensuring that users can efficiently locate available items and submit borrowing requests.

Week 7-9

After the core functionalities are developed, integration testing will begin. This phase will involve testing the interaction between the front-end and back-end, ensuring that all data flows correctly between the user interface and the database.

The testing will contain usability tests with end users to assess ease of use and identify areas for improvement.

Unit tests will be used for individual components to verify that each module functions as expected. Integration testing will verify the interaction between various modules, ensuring that the system works as a whole.

Week 9-10

The final phase will include a full system test, covering usability, unit, and integration tests. In addition, stress tests will be conducted to ensure that the system can handle many simultaneous users and transactions.

The documentation will be finalized, providing a clear guide for users and system maintainers. This will include user manuals, API documentation using OpenAPI specification, and a system architecture guide.

A final demo will be prepared for stakeholders to validate the system for deployment. After a final evaluation of the client, the system will be deployed to the production environment. In the end, a submission on Canvas will be made with the necessary deliverables.

8.1.2 Overview planning

To facilitate a structured and systematic approach to the design and implementation of the system, a Gantt chart was developed. This graph provides an overview of the planning

throughout the design and implementation process.

The first three weeks are dedicated to the requirement gathering phase; during this stage, weekly meetings are organized with stakeholders to gather and reaffirm the determined requirements.

In weeks two and three, the design phase starts, a Figma design is made and demonstrated to the stakeholders, and UML diagrams for the system are designed. The implementation of the back-end and database is scheduled to be conducted over weeks four to eight.

The front-end development is planned for weeks three to seven. The testing phase takes place during the entire development phase as the system is being implemented. However, it is mainly focused on the last four weeks, as at that time, integration tests can be performed and the system is closer to completion. The documentation is planned to be worked on throughout the module.

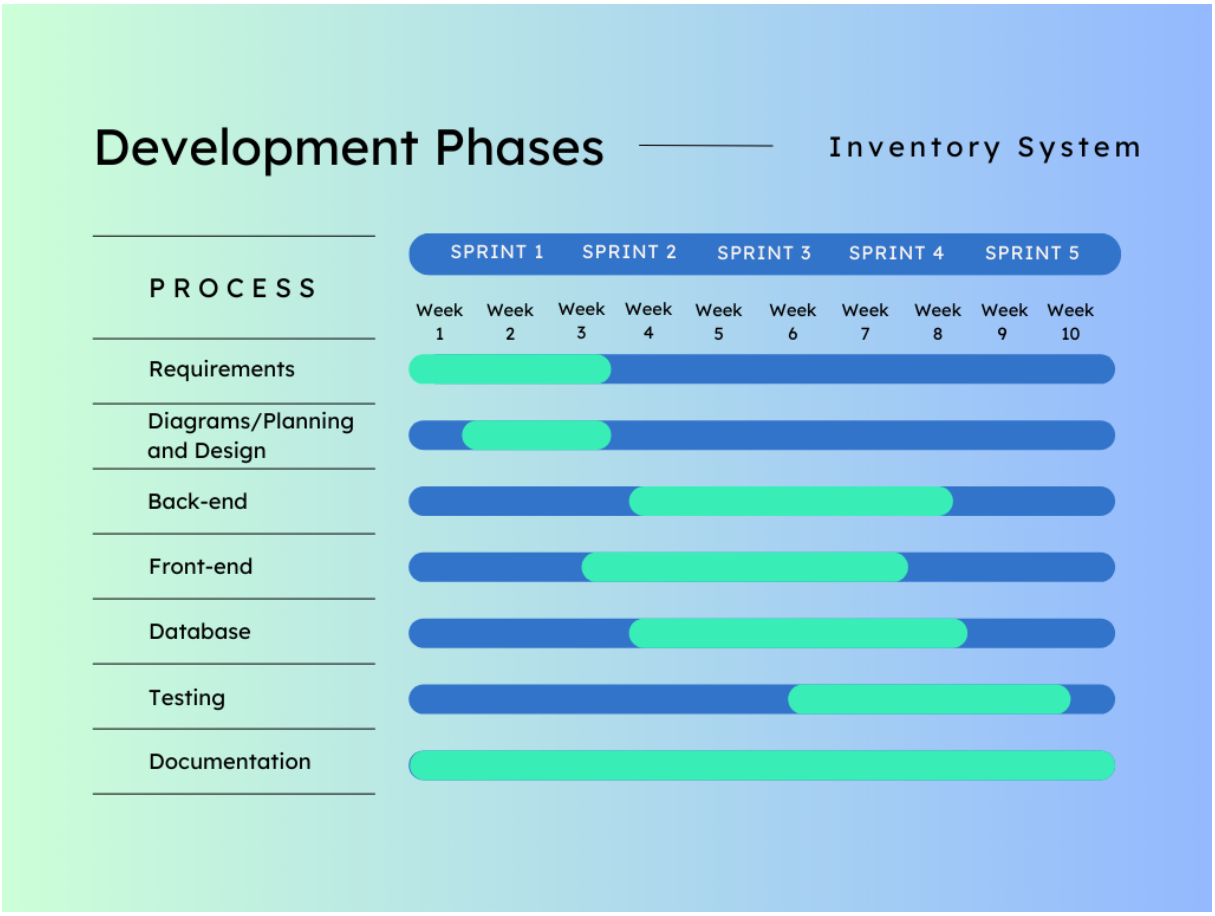
Furthermore, tools used for project management shall be discussed in this subsection. To manage task division and assignment, the development uses Trello. GitLab is chosen as the version control tool. The shared documents are managed within a shared Google Drive Folder, and WhatsApp is chosen as the main communication application.

General Planning Table

Weeks	**General Planning**
Week 1 *Feb 3-9*	• Interview with the stakeholder • Start gathering requirements • Start Figma prototype • Create UML diagrams • Documentation
Week 2 *Feb 10-16*	• Continue UML diagrams • Continue Figma prototype • Confirm requirements with stakeholders • Documentation

Weeks	General Planning
Week 3 *Feb 24-Mar 2*	• Continue Figma prototype • Prepare planning • Present project design to stakeholders • Documentation
Week 4 *Mar 3-9*	• Start back-end development • Start database development • Start front-end development • Documentation
Week 5 *Mar 10-16*	• Continue back-end, database, and front-end development • Documentation
Week 6 *Mar 17-23*	• Continue back-end, database, and front-end development • Documentation
Week 7 *Mar 24-30*	• Continue back-end, database, and front-end development • Complete database by visiting the Pervasive Systems laboratory • Documentation
Week 8 *Mar 31-Apr 6*	• Perform testing (usability, unit, system, API, integration tests) • Documentation
Week 9 *Apr 7-13*	• Perform testing • Documentation

Weeks	General Planning
Week 10 *Apr 14-20*	• Perform testing • Final presentations • Finish documentation



8.2 Risk Analysis

In this section several aspects of this project, which may result in unexpected behavior in the development process, will be illustrated. It will also describe how these risks can be minimized.

In this project that requires a working outcome, time management is a major priority. The final deliverable should be as complete as possible to meet the client's expectations, while ensuring sufficient clarity to facilitate a well-defined and satisfactory completion. To manage this risk, a planning is created at the start of the project, which describes for each week the dedicated part of the system. Additionally, the requirements are prioritized in the MoSCoW categories, to ensure a clear and well-defined scope of the project.

Another risk that may be faced in this project, is the risk of miscommunication with

the client. As the project idea is created by the client, it is important to make sure that they are satisfied with our solution to the problem and to manage their expectations to be in alignment with the progress. The client will be present throughout all the phases of the project, by explaining the problem they face, be part of the requirements gathering, and provide feedback throughout the design and implementation.To minimize delays in the implementation process caused by miscommunication, client meetings will be scheduled on a weekly basis. These meetings will make sure that the client stays up to date with the team's progress, and ensures that any misunderstanding will be identified and corrected within a week.

As for all inventory management systems, there are security risks involved. During the development of the system, we should make sure that the system is protected against vulnerabilities. This will be achieved by ensuring proper testing is done throughout the development process. Not only the functionalities of the system will be tested, but also security aspects of the application.

For their problem, the client expects a scalable system. When not done right from the beginning, this can result in scalability issues. Therefore, we will already be designing with scalability in mind, this will provide a solid foundation on implementing a scalable solution.

In order to host the website and the database that are being implemented on a sub-domain within the Pervasive System network, which will be accessible to the researchers and the doctoral students within the laboratory, an IP with *.eemcs.utwente.nl was requested from the ICT Servicedesk. The response was positive, indicating that as long as a laptop with ethernet can be provided, this address can be hosted. However, if the ICT desk experiences some security concerns or infrastructure limitations, the product deployment can experience delays and inaccessibility. Additionally, if the provided laptop experiences technical issues such as hardware failure, the hosted website would not be accessible at all times, which would interfere with the expected behaviour of the product. These risks can be minimized by communicating with the ICT desk about the scope of the project and ensuring that the chosen laptop is suitable for this task.

To provide a system that uses the university login system, as preferred by the client, we depend on another party within the University of Twente. This, combined with the lack of experience in the team in implementing this university login, could also pose a risk. To manage this risk, we will start by implementing and designing our own log-in system in which we will store users information in our own database. This will ensure that the system has a secure login system, after which we can look at implementing the university login system to make it even more secure and convenient for the users.

8.3 Reflection

8.3.1 Process

During the project we were expected to follow the design methods taught during our bachelor program. We decided to spend the first couple of weeks gathering requirements and creating a design for the system. The design phase of the project went smoothly as

we had the ability to put a lot of time and effort into creating the requirements (Section 2), diagrams and the prototype (Section 4) based on our first meeting with the supervisor. During these weeks we were also able to get clear feedback on everything we made, which we were able to swiftly use to improve our design. However, after week 4 the work load increased as reflection part of the module also started demanding more attention from us. Combined with the freedom to decide how much progress we made between meetings with our supervisor, it meant the focus was sometimes put more on the reflection part than the project. As the reflection component only ended (excluding the resit) a couple of weeks before the end of the module, it was difficult balancing this work load with the project throughout most of the module. Both required a lot of attention and it sometimes felt we needed to give more than we had.

We decided to work with a structure where the development for front-end and back-end were separated. For most in our group this was a new way of working, as during previous modules where working as a full-stack developer was more normal. In this we decided to put 2 people on front-end and 4 people on the back-end (including database). This had the benefit that tasks were more easily divided and that communication withing the groups was easier, although bigger design choices would still be made with the group as a whole and communication between the two "teams" was frequent. During the module it became apparent that it was important that both the front-end and back-end would regularly inform each other of their needs and limitations as otherwise time would be lost.

8.3.2 Challenges

During the module, several challenges were encountered. One of the most prevalent during the entire module was the lack of leadership. Our group did not have a clear leader that would make decisions, most members of our group did not really have strong opinions about decisions that had to be made. An example of this is the back-end system we are using, we spend a couple of weeks going around in circles of people saying they did not really care about what we would choose to use for the back-end. Even the decision that we were going to use Java as the scripting language for the back-end took multiple discussions.

Another challenge was making sure enough work was put in throughout the module. With the full freedom of deciding how much time we spend on the project, and even on which part of the project, we fell into the common pitfall of ignoring the report for too long which meant that we were put under more time pressure for the report than was necessary if we had started on time.

Lastly, the IMS development process was slowed down as a result of waiting on answers for certain requests for the different departments in University of Twente. The development team has

8.4 Contributions

8.4.1 Report and Presentations

Section	Melania Vartic	Jorim Hebbink	Duong Thu Huyen	Luuk Alfing	Nilufer Guldali	Ruben Hannink
Context	100%	0%	0%	0%	0%	0%
Requirements gathering	0%	0%	0%	100%	0%	0%
Architectural Design	90%	10%	0%	0%	0%	0%
Mock-up and prototype	0%	0%	0%	0%	0%	100%
Implementation and detailed design	40%	10%	0%	20%	15%	15%
Testing	20%	0%	80%	0%	0%	0%
Security	0%	10%	0%	90%	0%	0%
Discussion and Process	50%	0%	0%	0%	0%	50%
Conclusion and Recommendations	0%	0%	0%	0%	100%	0%
Figma mock-up	0%	0%	0%	0%	0%	100%
User Manual	90%	0%	0%	0%	0%	10%
Technical Manual	0%	90%	0%	0%	0%	10%
Presentation	40%	0%	0%	0%	0%	60%

8.4.2 Front-end

Item	Nilufer Guldali	Ruben Hannink
Inventory pages	0%	100%
Login/Signup pages	95%	5%
Category pages	0%	100%
User pages	100%	0%
Projects page	0%	100%
Create Item page	0%	100%
Supervision pages	0%	100%
Checkout page	0%	100%
Home page	100%	0%
Error page	0%	100%
Alert system	0%	100%
ApiRequestData system	0%	100%
Modal system	0%	100%
Session system	0%	100%
Navigation helpers	0%	100%
Date converters	25%	75%
Style sheets	0%	100%

8.4.3 Back-end

Item	Melania Vartic	Jorim Hebbink	Duong Thu Huyen	Luuk Alfing
Configuration	0%	100%	0%	0%
Database Schema and Config	0%	100%	0%	0%
User logic	0%	0%	0%	100%
Borrow logic	0%	0%	100%	0%
Borrow Requesting logic	0%	0%	100%	0%
Item Collection logic	100%	0%	0%	0%
Item logic	100%	0%	0%	0%
Category logic	0%	100%	0%	0%
Variable logic	20%	80%	0%	0%
Role logic	0%	20%	0%	80%
Authentication logic	0%	0%	0%	100%
Authorization logic	0%	0%	0%	100%
Damage Report logic	100%	0%	0%	0%
Supervisor logic	0%	0%	0%	100%
Type Variable logic	20%	80%	0%	0%
Exception Handling	20%	40%	20%	20%
Projects logic	100%	0%	0%	0%
Email logic	0%	100%	0%	0%
Calendar logic	0%	100%	0%	0%
Filtering logic	80%	20%	0%	0%
Unix Time converter	0%	100%	0%	0%

8.5 Recommendations

Although the current version of the system is functional and extensible, there are always ways in which a system can be improved. Some recommendations for future improvements are integration with the UT Microsoft Authentication, mailbox integration, and hosting the website.

8.5.1 Integration with UT Login

Due to time limitations and security concerns, integration with the University of Twente login portal has been deemed unfeasible. Firstly, communication with the IT department of University of Twente has been rather slow, thereby making the integration with the UT login task more difficult. Moreover, in a later stage of development, the department informed that it is a university policy not to connect university systems to systems developed by students. Therefore, it was decided that the development team would implement an independent authentication system. This separate authentication approach has worked for the Inventory Management System and integrated well. However, it would be a point of recommendation to integrate the university login system to the project. With university authentication, system users could open their inventory account directly, and this would have an advantage of not having to remember two separate passwords for these systems.

8.5.2 Hosting the Website

Another topic of recommendation would be hosting the inventory system within the University of Twente infrastructure, more specifically, under Pervasive Systems network. This option has been explored extensively and numerous emails of communication were exchanged between the development team and the corresponding university departments. The team has requested a subdomain to host the website and its database. In this way, the application would be accessible to researchers and employees of Pervasive Systems. In the request, plans for deployment were also included. However, due to logistical hindrances, there were delays in acquiring the required MAC address and other necessary information. Therefore, the process could not be completed within the project time frame.

8.5.3 Mailbox

It was originally designed that the Inventory Management System would have a functional mailbox that would send notifications. These notifications include approaching return dates and confirmation of actions performed within the system. Additionally, the authentication manner such as one time password and reset password functionalities are also made possible in this way. During development, the team used a mailbox hosted on a personal raspberry pi of one of the team members, which is why the development team requested a dedicated functional mailbox for the IMS instead.

After requesting a functional mailbox from the corresponding departments, access was given to a mailbox to be used by the project. However, as the application sends emails using SMTP, password is required to authenticate with the server. This made it unfeasible to integrate the provided mailbox with the Inventory Management System as it could only be accessed via personal university accounts. A request was made to acquire credentials to authenticate with the SMTP server, however, this request was unanswered. Therefore, currently the system is not connected to a functional mailbox.

It should be noted that the email functionalities are implemented, and once the credentials to authenticate with the SMTP server are provided, the system should integrate seamlessly with the email service. Therefore, integrating a proper mailbox remains a future recommendation.

Chapter 9

Conclusion

In this report, we have explored various aspects of the Inventory Management System developed for the Pervasive Systems Laboratory. These aspects include the development process, testing strategy, and design process throughout the module. Initially, the system requirements were gathered, and put into an order of priority. Based on these requirements, the design phase began. It should be mentioned that the design process continued throughout the module, as the implementation progressed and new requirements emerged, while some requirements were removed due to limited resources or access (such as the integration of Microsoft UT authentication). Then, a Figma prototype was made and presented to the stakeholders, giving an insight into the initial design of the system. Afterwards, the implementation phase continued iteratively with incorporation of feedback and new insights, until the end of the module. Testing took place alongside the implementation, and became more intensive after major functionalities were implemented.

The objective of this project was to accommodate and ease the process of borrowing inventory items within the Pervasive Systems Laboratory. Our solution, an account-based Inventory Management System achieves this by providing numerous functionalities to manage inventory items, supporting multiple type of items, enabling users with higher privileges to make additions to the inventory. Thus, enabling them to expand the inventory, and by doing so, making the system scalable.

With the use of technologies Java Spring Boot, Bootstrap CSS, React, Hibernate and MariaDB, it was possible to implement the Inventory Management System. In order to ensure maintainability, a developer manual has been included within this report, along with a user manual that was prepared to improve usability.

Chapter A

Figma mock-up

Images of the latest version of the Figma mock-up. Includes all pages that were created in the mock-up. All mock-up pages were intended to show the interface when the user is logged in as an admin, as this role has the most permissions. Created using the Bootstrap 5 UI Kit by Jitu Chauhan.

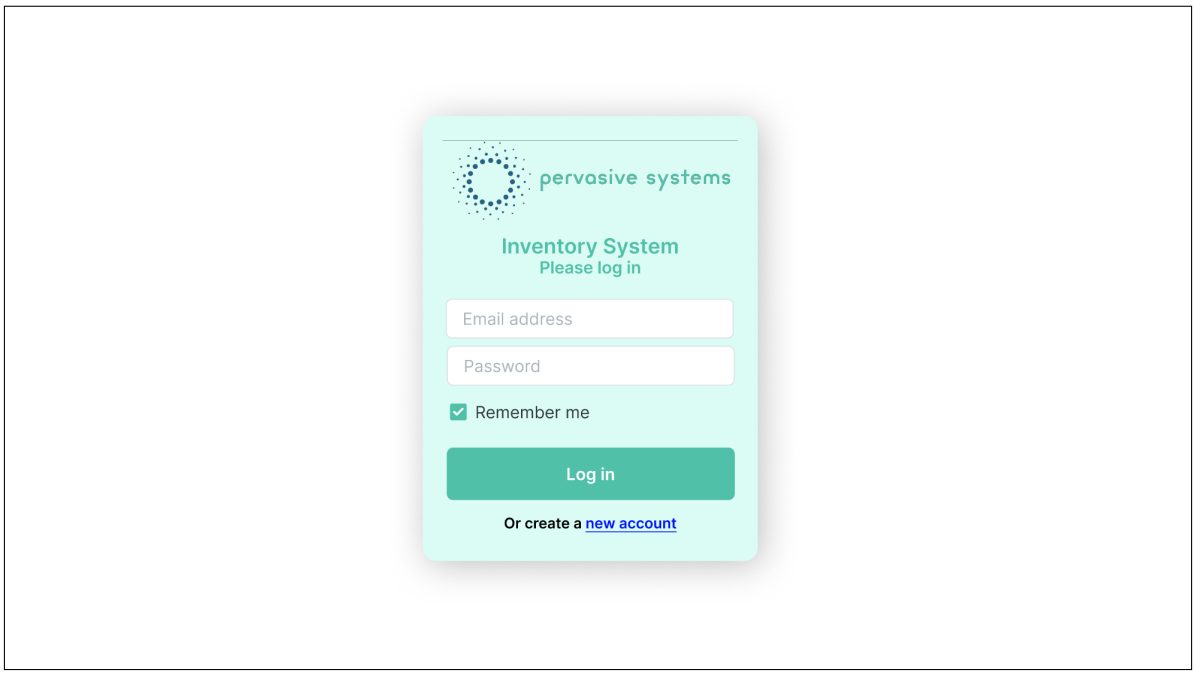


Figure A.1: Mock-up of the login page.

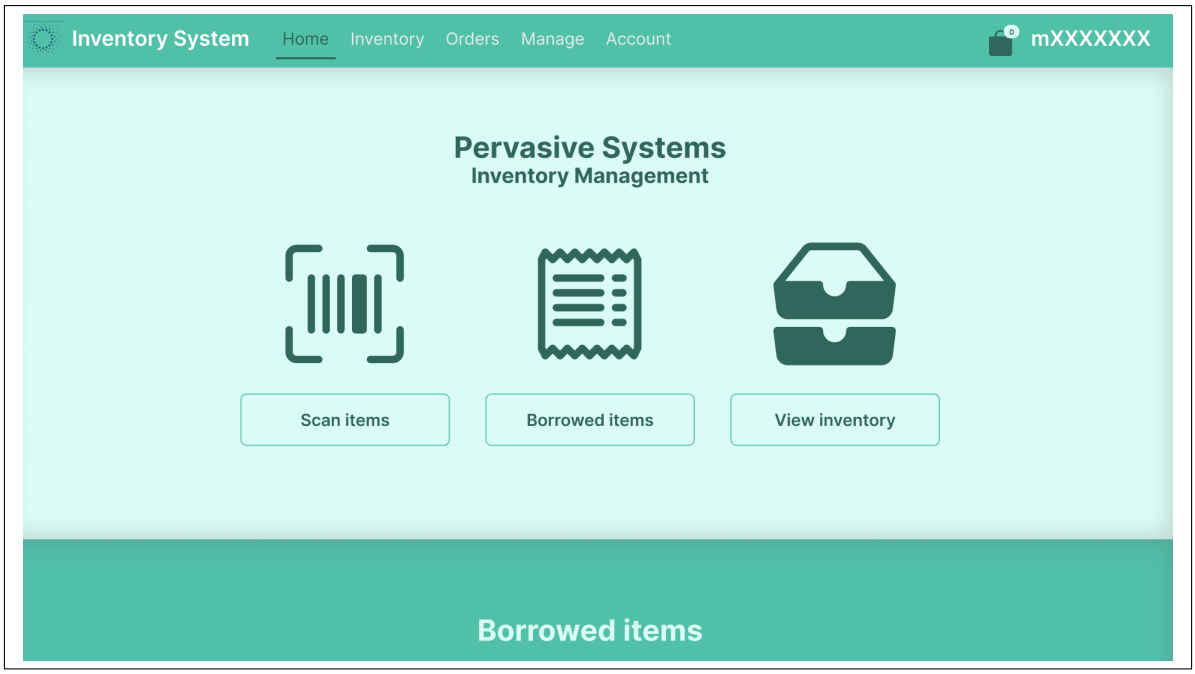


Figure A.2: Mock-up of the home page.

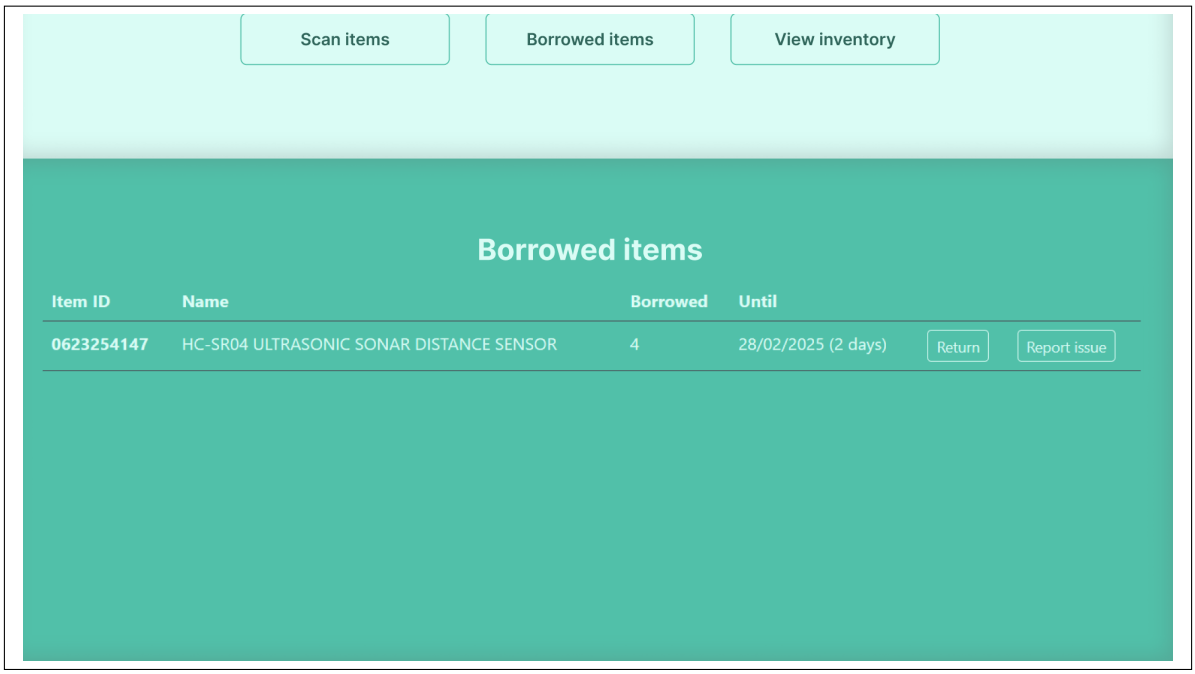


Figure A.3: Mock-up of the home page, scrolled down to show the borrowed items table.

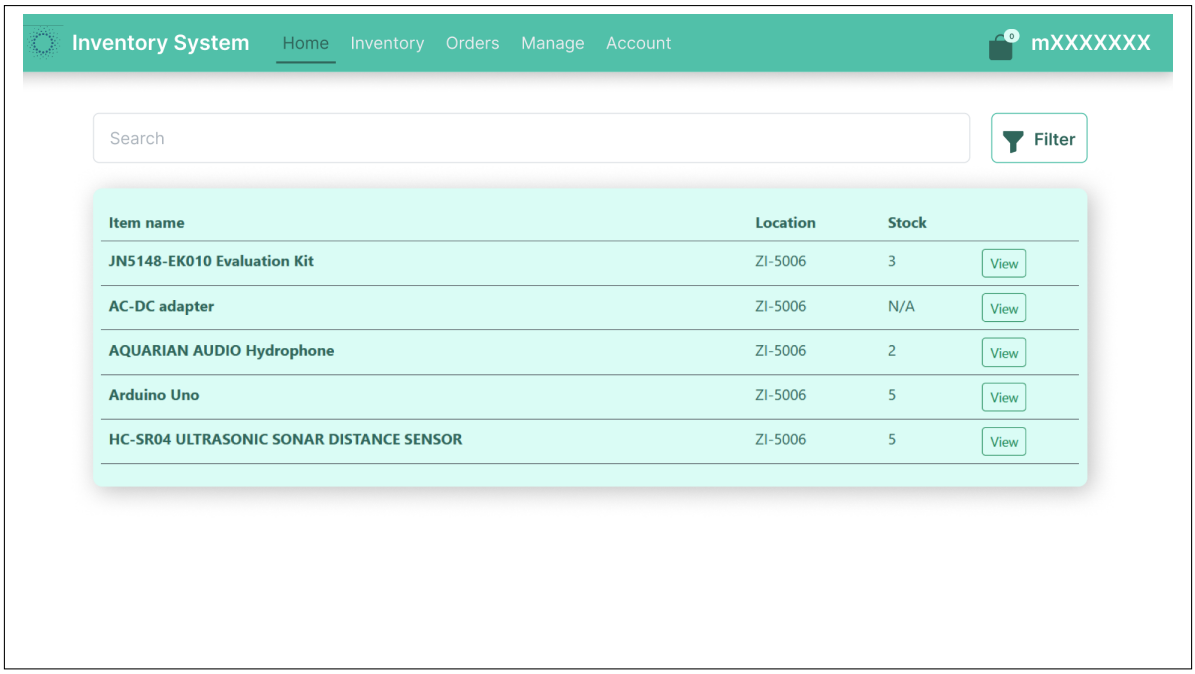


Figure A.4: Mock-up of the inventory page.

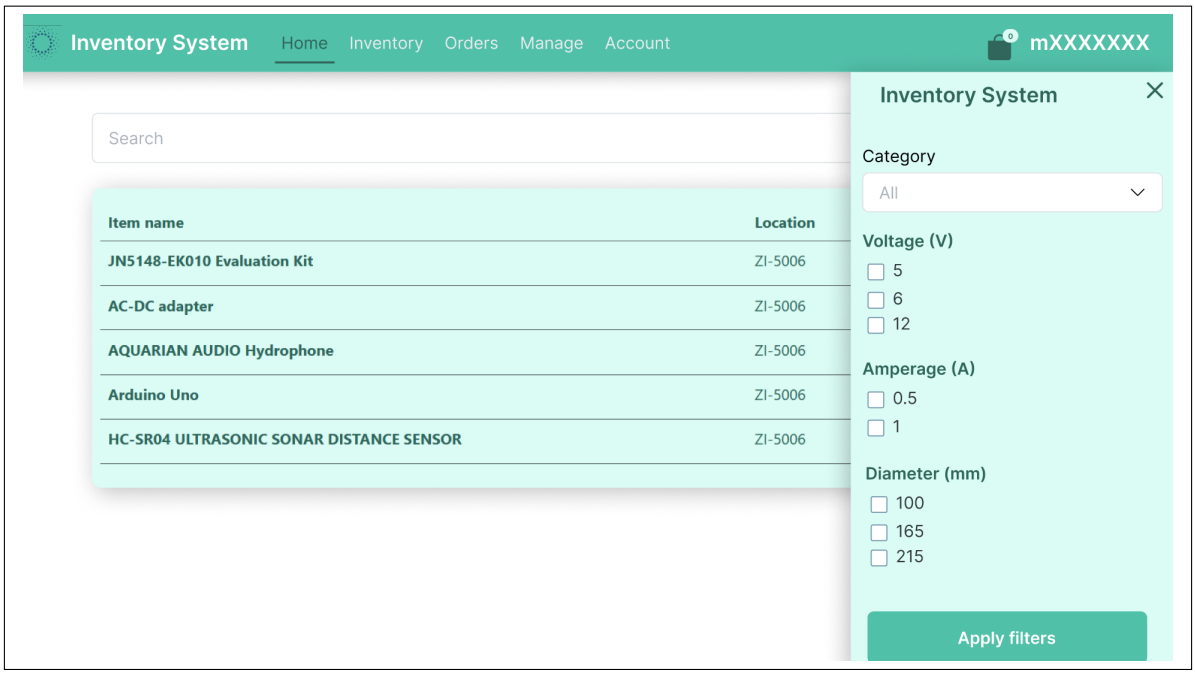


Figure A.5: Mock-up of the inventory page, with the filtering tab open.

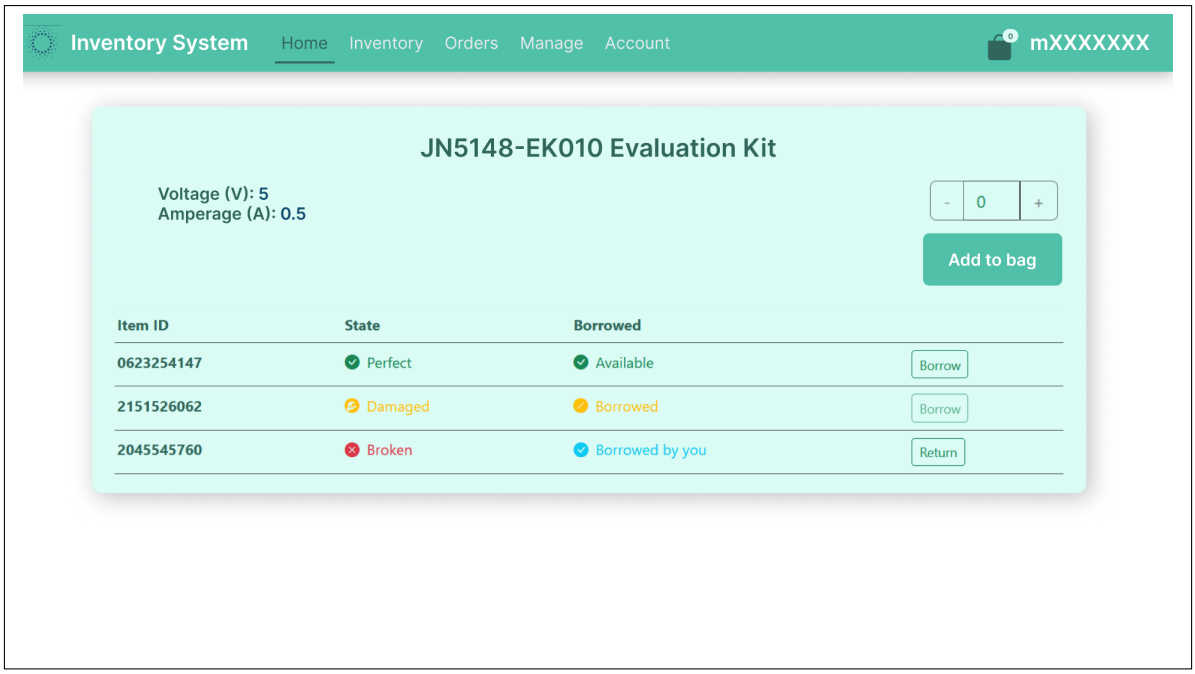


Figure A.6: Mock-up of an item page. Showing a unique item collection.

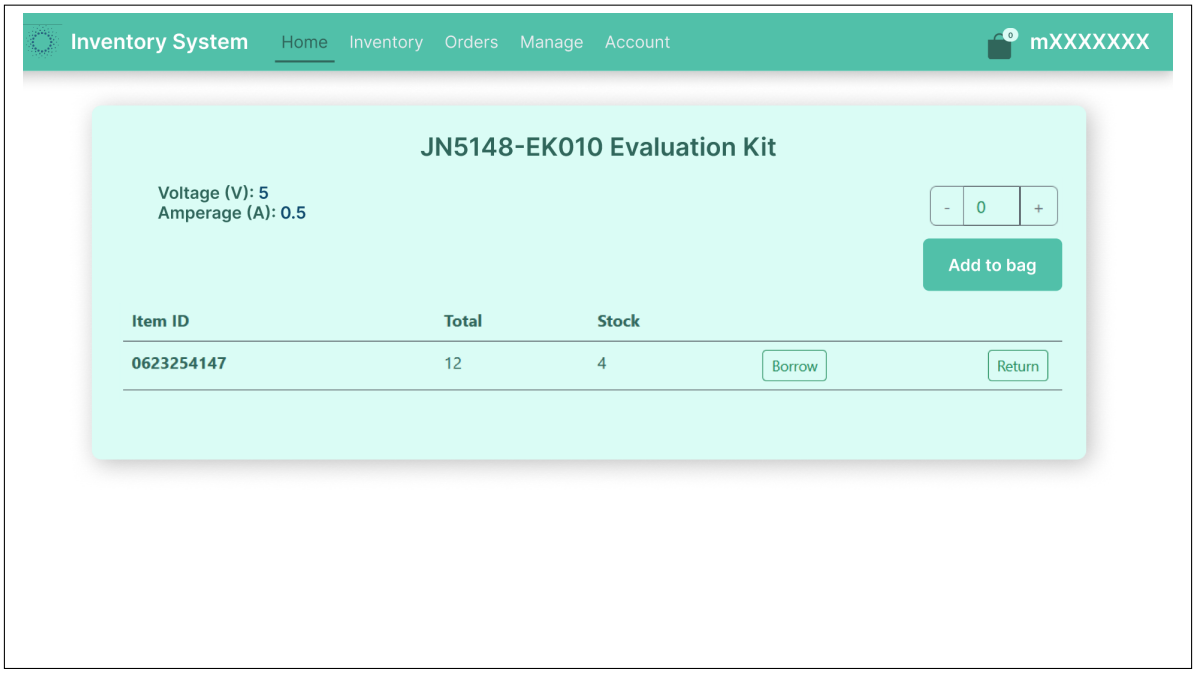


Figure A.7: Mock-up of an item page. Showing a shared (bulk or infinite) item collection.

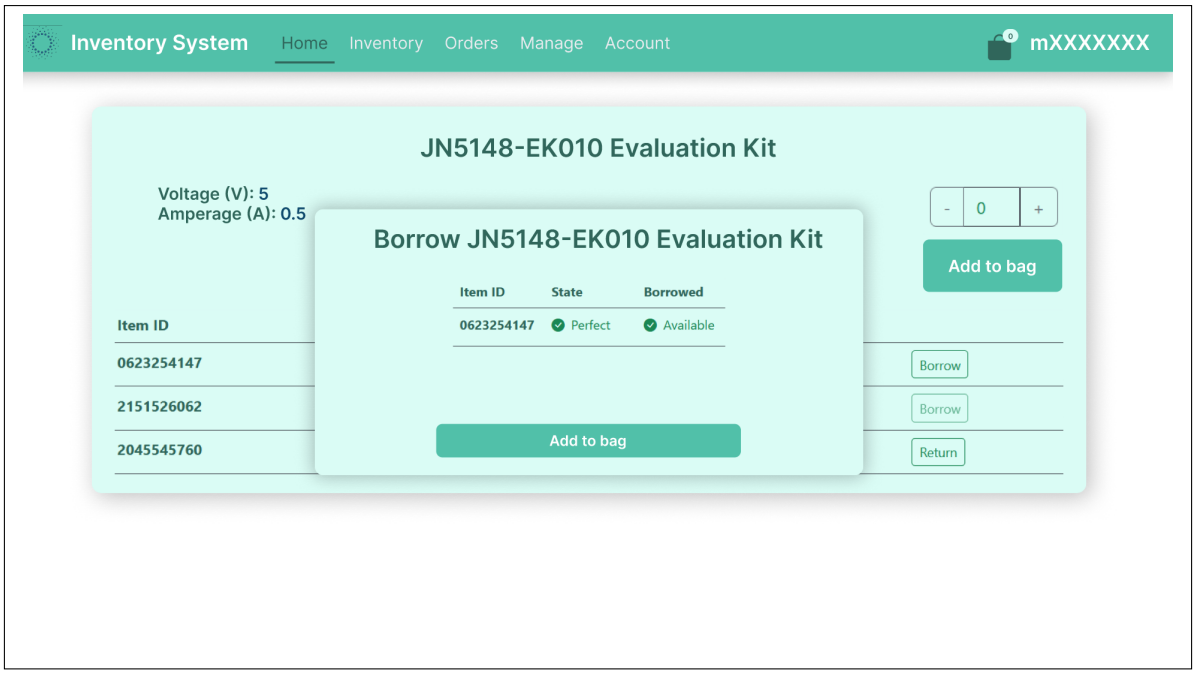


Figure A.8: Mock-up of an item page, with the confirmation prompt to borrow an item.

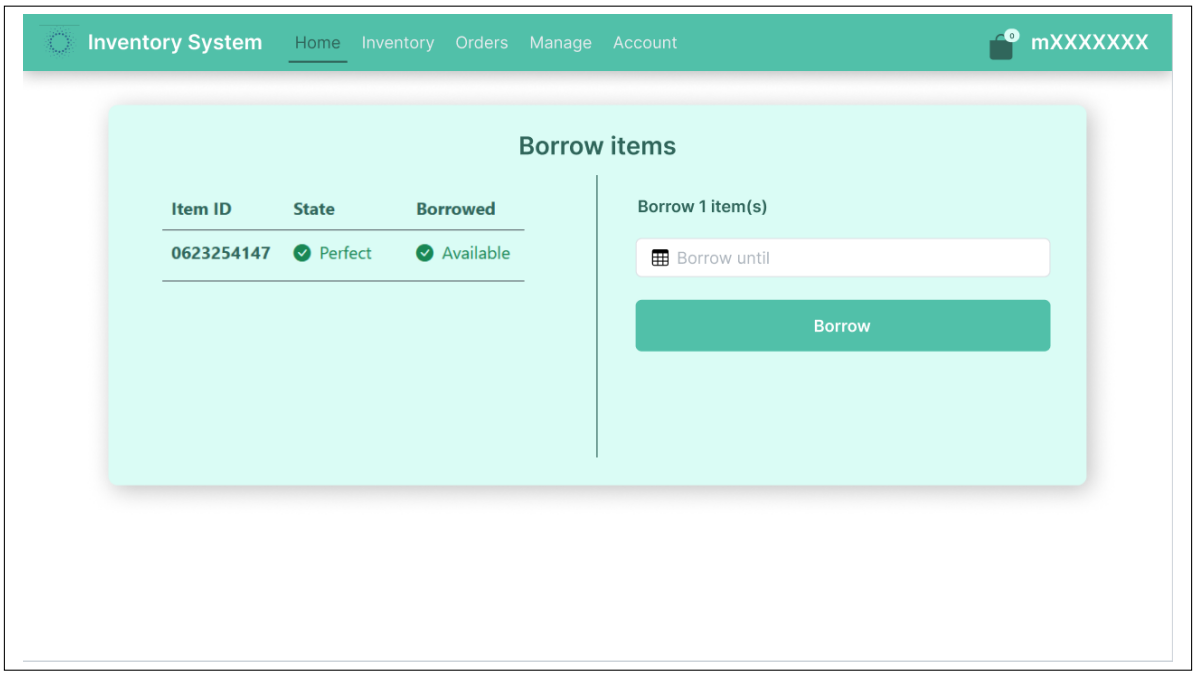


Figure A.9: Mock-up of the checkout page.

Chapter B

User Manual

The user manual is split up into multiple sections based on the different roles within the system (Guest, Employee, and Administrator). Additionally, a general manual is provided for things that are the same for all roles.

B.1 General Manual

The general manual focuses on the actions the you can take while you are logged out of the system. Whenever opening the website while not logged in, you will be presented by the login screen first.

B.1.1 Make an account

1. When you go to the website for the first time, pick the `new account` option to register on the application

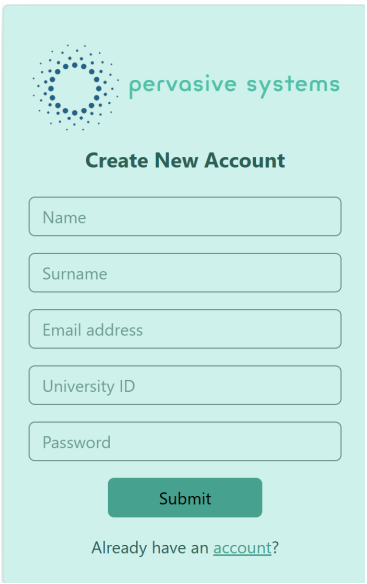


Figure B.1: Signup page

2. Fill in your first and last name
3. Fill in your email address issued by the university (ending in `@student/employee.utwente.nl`)
4. Fill in your university number (sXXXXXX or mXXXXXX)
5. Fill in desired password, must be at least 8 characters long

6. Click on `Submit` button
7. After signing up you will be redirected to the login page

B.1.2 Login

There are two different ways you can log into the system if you have an account.

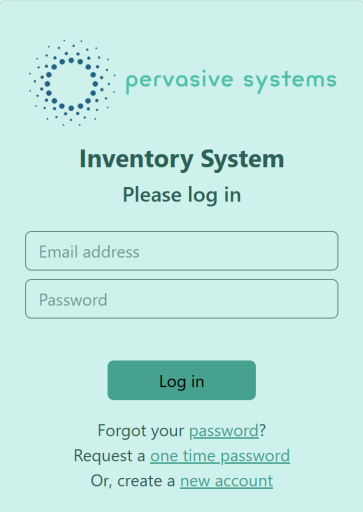


Figure B.2: Login page

Password

The first option is using the password you created during the sign-up phase.

1. Provide the email of your registered account
2. Provide your password
3. Press `Log in` button

One-Time-Password

Another option is using the one-time-password. This can be chosen from the login page. Note that if it takes more than five minutes to login with the OTP, you will be required to request a new OTP.

1. Press on`OTP` button
2. Enter your email address
3. Check both your junk and main inbox for an email from ps-inventory-system

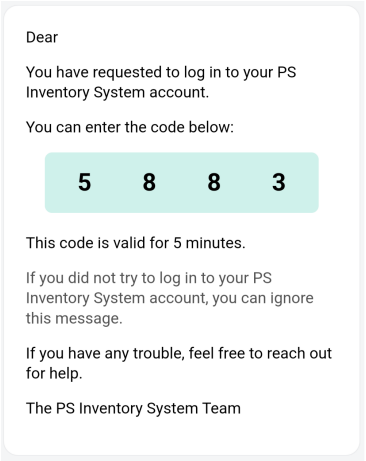


Figure B.3: The OTP email should look like this

4. Type the code into `One Time Password` field
5. Press`Log in` button

B.1.3 Reset Password

For resetting your password you start at the login page.

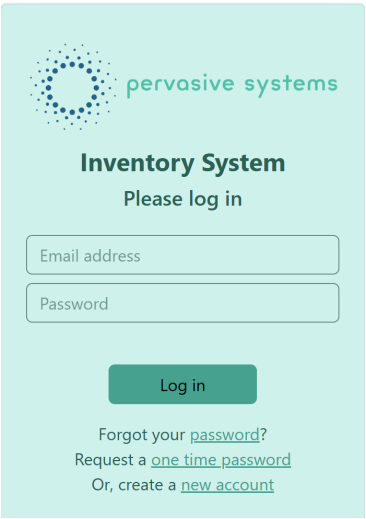


Figure B.4: Login page

1. Press on the password section of Forgot your Password button
2. Type your email
3. Check both your main and junk inbox for an email from the ps-inventory-system

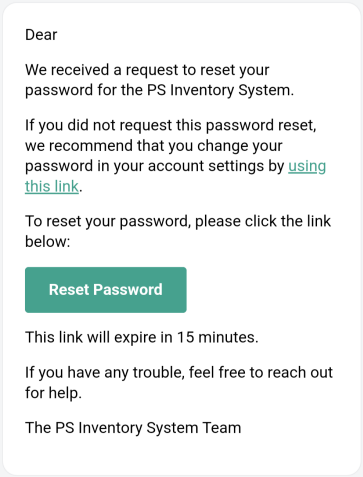


Figure B.5: Reset password email

4. Click on the Reset Password Button
5. Provide your new password

B.2 Guest Manual

B.2.1 Request Supervisor

1. Navigate to the Supervision page on the navigation bar.

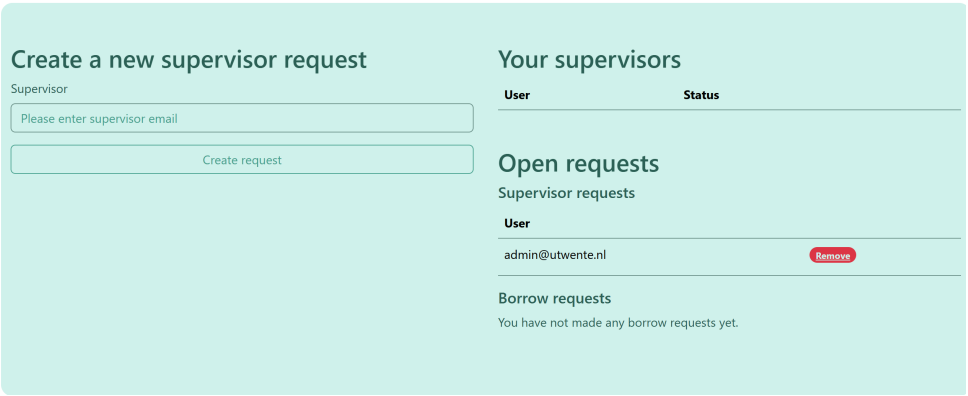


Figure B.6: Request supervisor email

2. Type your supervisor's email.
3. Click on create request button.
4. The request will appear as an open request similar to the one to `admin@utwente.nl` from the example.
5. Contact your supervisor to approve the request of supervision to be allowed to borrow items

B.2.2 Borrow an Item

1. Navigate to the Inventory page on the navigation bar.
2. Click on the item you wish to borrow.
3. Check the availability and condition of the items.
4. If the item is available and in good condition, click the add to the bag button.
5. Review the item once more in the pop-up.
6. Click the add to bag button.

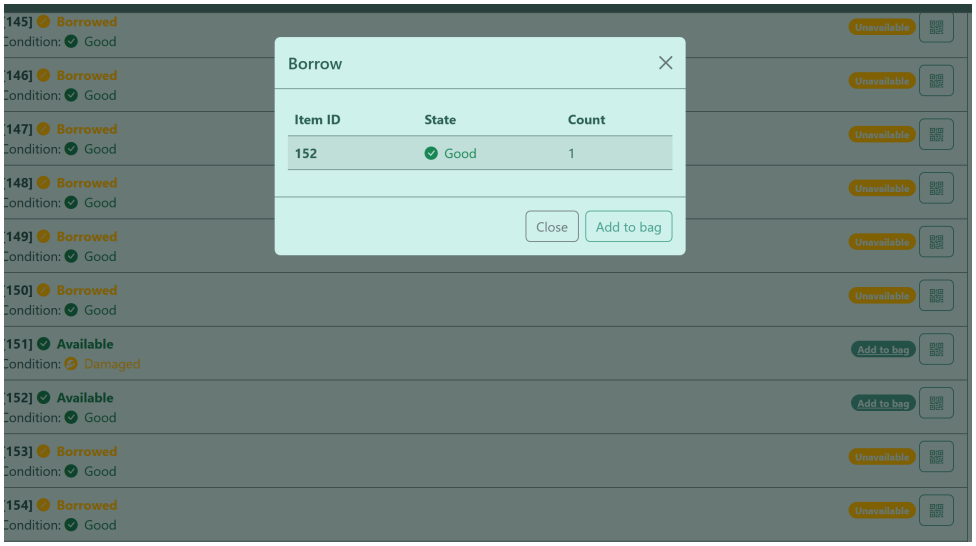


Figure B.7: Add to bag

7. Click on the bag icon located on the header to navigate to the checkout page.
8. Declare a date until when you would like to borrow the items.
9. Select your supervisor.
10. Click on the create request for borrow button.
11. If your supervisor accepts your request, this item is added to your borrowed items.

B.3 Employee/Supervisor manual

B.3.1 Borrow an Item

1. Navigate to the Inventory page on the navigation bar.
2. Click on the item you wish to borrow.
3. Check the availability and condition of the items.
4. If the item is available and in good condition, click the add to bag button.

5. Review the item once more in the pop-up and click the add to bag button.
6. Click on the bag icon located on the header to navigate to checkout page.
7. Declare a date the item will be borrowed until.
8. Click on borrow all items button.

B.3.2 Approve/Deny a guest borrow request

1. Navigate to the supervision page on the navigation bar.

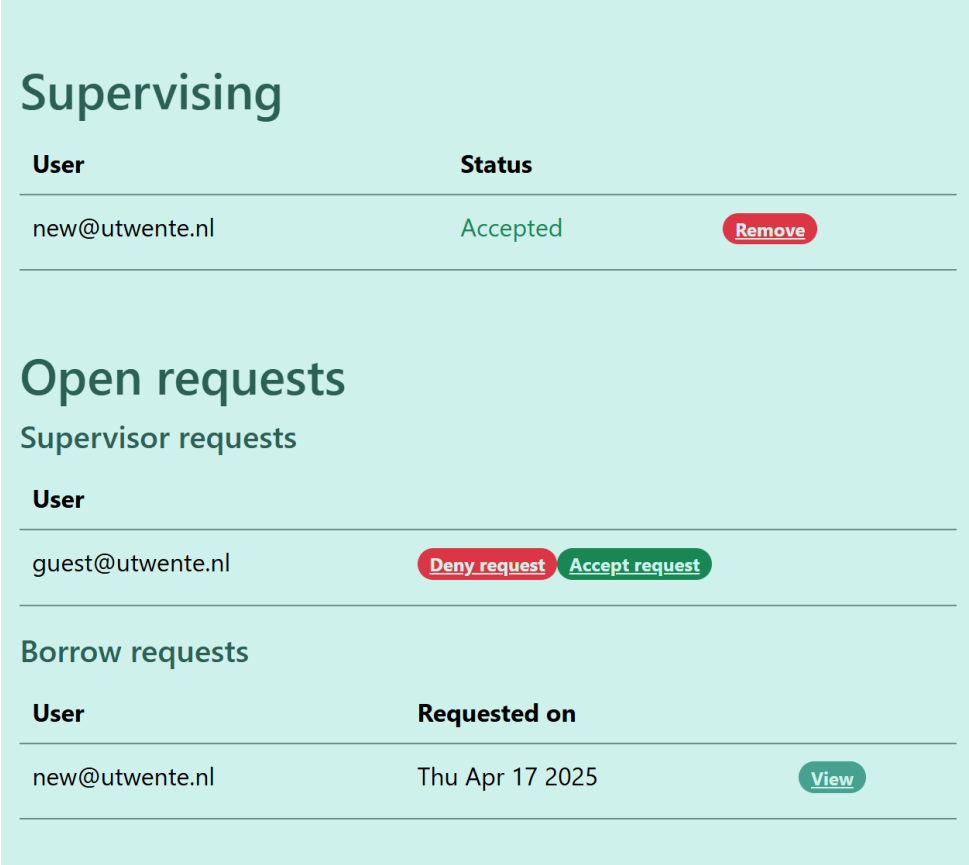


Figure B.8: Supervisor Page

2. At the bottom of the page, a list with the title `Borrow requests` is located.
3. View the emails of the users who made the borrow requests and the date that request was made.
4. Choose a borrow request and click view button.
5. Here view name, amount, and borrow until date of the item.
6. You can deny or accept the request to borrow by clicking on the corresponding buttons.

B.3.3 Add a Collection of Items

1. Navigate to `Manage`
2. Press on `Create item`

Figure B.9: New Collection Page

3. Choose name of item collection
4. Choose a description (optional)
5. Choose a vendor (optional)
6. Choose a room (optional)
7. Choose a position of item in the room (optional)
8. Choose collection type from drop down list (press the information button next to the type to see an explanation of the collection types
9. Choose an existing category
10. Choose an existing project (optional)
11. Give a number of items
12. Set value for each variable of the category

B.3.4 Search and Add Projects

1. Navigate to Manage projects from the `Manage` drop-down on the navigation bar.
2. Search for the project name you wish to see.

3. If the project does not already exist, click on `Create` button next to the search bar.

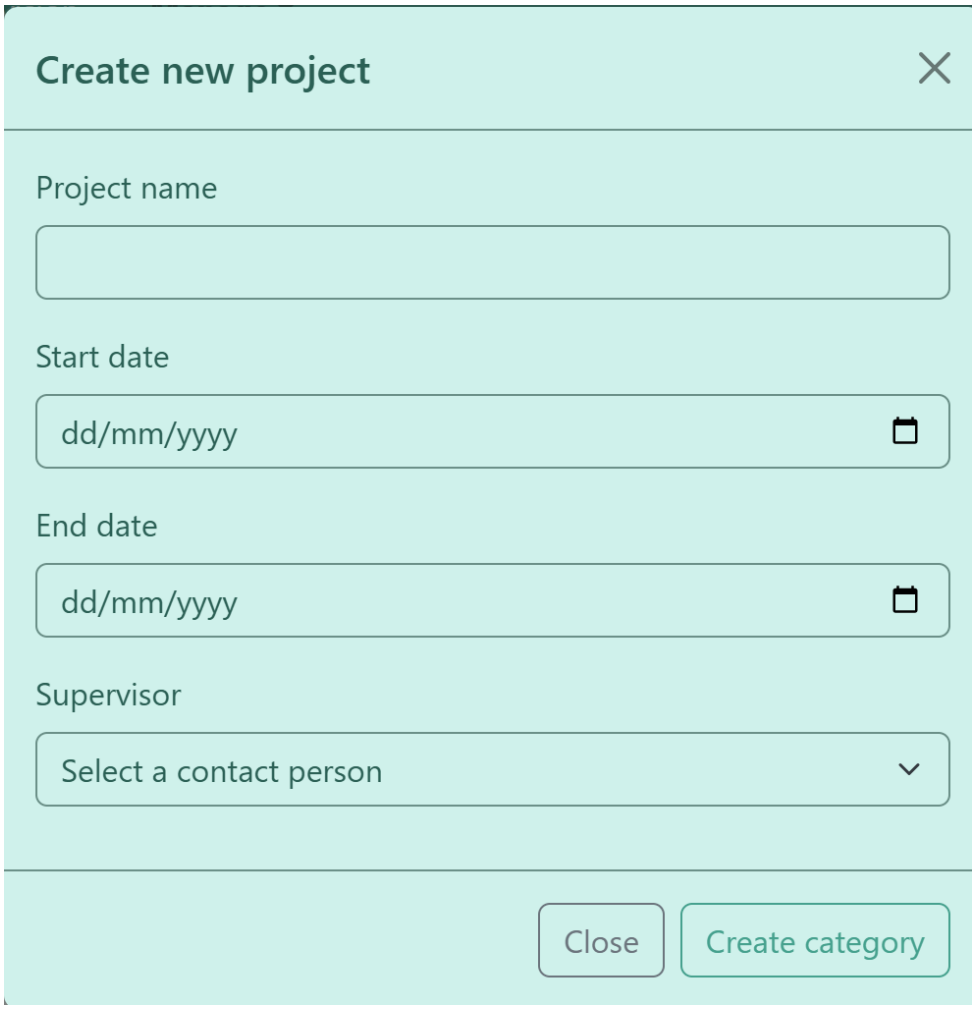


Figure B.10: Project Page

4. Provide the name, start and end date of the project.
5. Select a contact person for the project from the drop-down menu.
6. Click on `Create Project` button.

B.3.5 Delete projects

1. Navigate to `Manage projects` from the `Manage` drop-down on the navigation bar.
2. Search for the project name you wish to delete.
3. Click on view button.
4. Click on `delete project` button.
5. You will receive a confirmation message. Click confirm.

B.4 Administrator manual

B.4.1 Update or Delete a Collection

1. Navigate to `Inventory` either through the navigation bar or the `View Inventory` button
2. Search for the collection that you want to update/delete
3. Press the `View` button on that collection

Figure B.11: Collection Page

4. If you wish to delete it press the red button with a bin icon
5. If you wish to edit it press the button with a pencil icon

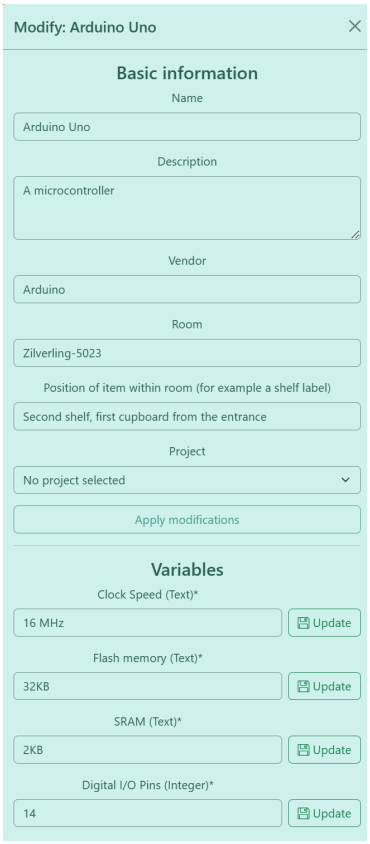


Figure B.12: Edit Collection Page

6. Change whatever field you want and press the `update` icon

B.4.2 Add a Category

1. Navigate to `Manage`
2. Press on `Manage Categories`
3. Press on + sign next to the search bar

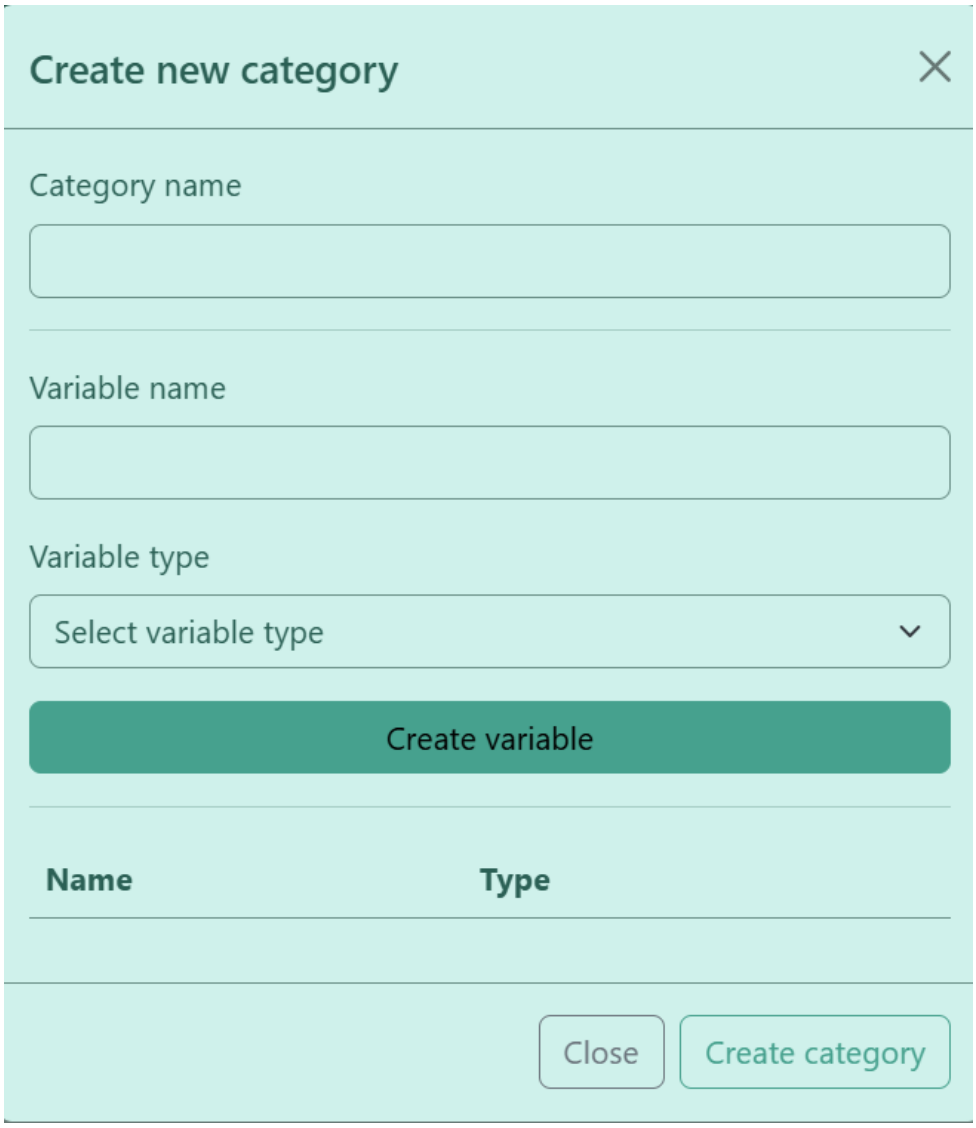


Figure B.13: New category Page

4. Choose name of category
5. Now you can add variables to the category
6. Choose name of variable
7. Select type from drop down list
8. Press on `Create Variable` button
9. Repeat last 3 steps as many times as you want to add more variables
10. Click on `Create category button`

B.4.3 Add variables to a category

1. Navigate to `Manage`
2. Press on `Manage Categories`

3. Press on `View` button on the desired category

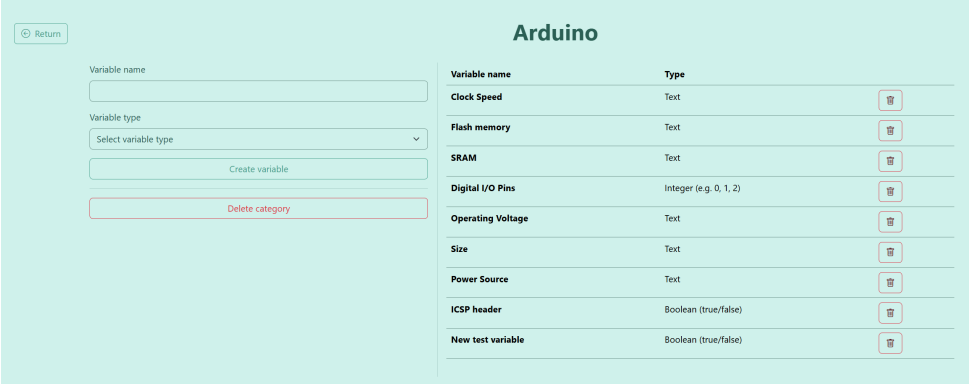


Figure B.14: New variable Page

4. Choose name of variable
5. Select type from drop down list
6. Press on `Create Variable` button
7. Repeat last 3 steps as many times as you want to add more variables

B.4.4 Change the role of a user

1. Navigate to the `Manage users` from the Manage drop-down on the navigation bar.
2. Search for the user you wish to change the role of by pressing `filter by name`.
3. In the role column, click on the current role of the user to enable drop-down.
4. Select the new role of the user.
5. Click on `change` button just next to the role.

B.4.5 Delete a user

`Note:` this action is irreversible.

1. Navigate to the `Manage users` from the `Manage` drop-down on the navigation bar.
2. Search for the user you wish to change the role of by pressing `filter by name`.
3. In the borrower information column, click on `Borrower information` button.
4. Scroll down to the bottom of the pop-up.
5. Click on `Warning! Delete user from the system` button.

Chapter C

Technical Manual

C.1 Introduction

The Inventory Management System (IMS) was created as a Bachelor Design project in 2025 for the Pervasive Systems research group (PS) at the University of Twente. The goal of the IMS was to organize and keep track of all the (hardware) items PS had in their storage rooms and who borrowed items. As a result the PS Inventory System has been created as a web-based application build using the Java Spring Boot framework.

C.1.1 Used Technologies

The project makes use of a variety of libraries and frameworks for both the client and server application.

Client

The client is build on top of the JavaScript React framework and makes use of Bootstrap for styling.

Server

The server's codebase is created for Java 23 and build on top of the Spring which made creating all different components easier. Using the framework we could easily combine the Hibernate database entities and repositories with custom Data Transfer Objects (DTO), Spring services and REST Controllers.

Email

The sending of emails consists of two parts, the email HTML templates using ThymeLeaf as template processor allowing to set certain values like user names, and other changes to the HTML on runtime. If a template is processed it is sent to the specified user by the JavaMail sender.

C.2 Getting Started

C.2.1 Prerequisites

Java 23 JDK

The application is created for the Java 23 JDK which is one of the most recent Java versions. The system has not been tested on any other JDK version. The JDK that was

used during development was OpenJDK-23.

NPM

The Node Package Manager (NPM) is used by the frontend to install dependencies and build the code. See https://docs.npmjs.com/downloading-and-installing-node-js-and-npm/ for more information about Installing NPM.

MariaDB Database

The project needs a database to store data, without it it is impossible to use the system. The default database the server uses is MariaDB, however it would also be possible to use a MySQL database since they are similar.

Any other database server is not supported since we cannot guarantee that the database schema file is valid as it is specifically created for MariaDB servers.

MySQL Database

To be able to use a MySQL database successfully the database driver needs to be changed to the MySQL driver. The following needs to be added to the configuration in `/config/application-default.properties` once the server is setup:

```
1 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

SMTP Email Server

For the system to be able to send emails, it needs to have valid credentials for an SMTP server. This can be from Microsoft, Gmail or any other email provider as long as it has an SMTP server.

Maven

For the development of the system Apache Maven is used for dependencies. If you want to manually install the project maven needs to be installed. You can install Maven here https://maven.apache.org/install.html.

C.2.2 Java Archive

If you want to use the project without having to build it yourself you can download and run the JAR directly. To start the server the following terminal command can be used:

```
java -jar InventorySystem.jar
```

After the server has started for the first time, it will immediately terminate with the message “Change the config credentials before running the application”. There should now be a folder called `config` with the file `application-default.properties` inside it. In this file you need to set the correct credentials for the Database and SMTP server after which you can save the file and restart te server, which should now run if the credentials are valid. If there are problems with the database connection the server will crash. If the email settings are invalid, an exception will be shown in the terminal but the system will continue to run.

Accessing the application

By default the system can be accessed on `http://localhost:8080/`. The port can be changed in the `config/application-default.properties` by changing the value of `server.port`.

Using HTTPS

By default the server does not make use of HTTPS since this requires a SSL/TLS configuration. If you have access to SSL/TLS credentials you can enable HTTPS by following the steps below: - The key should be in the PKCS12, this is required by spring boot. Add the following lines to the end of the `config/application-default.properties` file:

```
# Use HTTPS
security.require-ssl=true
server.ssl.key-store=/path/to/your/key.p12
server.ssl.key-store-password=certificatepassword
server.ssl.keyStoreType=PKCS12
```

And set `server.port=443` to use the default HTTPS port. Restart the server and you should now be able to access the server only using HTTPS.

C.2.3 Manual Installation

For the manual install, you have to build the JAR yourself following the steps below.

1. Clone the Project

First, clone the GitLab project or download the ZIP file containing the source code and extract it.

2. Open the Project

Open the project in your favorite Java IDE. During development, IntelliJ has been used and multiple run configurations have been created.

3. Install Dependencies

With the project open in the IDE, all Maven and NPM dependencies need to be installed.

- For IntelliJ: Run the `install` run configuration; this will install all dependencies.
- For other IDEs:
  - Install the Maven dependencies: `mvn install`
  - Build the client:
    - Open to the client folder: `cd src/main/client`
    - Install the dependencies: `npm install --include=dev`
  - Build the email templates:
    - Open the email folder: `cd src/main/email`
    - Install the dependencies: `npm install`

4. Build the Client and Email Templates

With all dependencies installed, it is now possible to build the client and email templates.

- For IntelliJ, this is automatically done when running the "Backend with Integrated Frontend" run configuration, which also starts the project.
- For other IDEs or if you do not want to build and run with IntelliJ:
  - Build the client:
    - Open to the client folder: `cd src/main/client`
    - Build the client: `npm run build`
    - The client build result is placed in `src/main/resources/static`
  - Build the email templates:
    - Open the email folder: `cd src/main/email`
    - Build the email templates: `npm run build`
    - The email templates are now placed in `src/main/resources/email-templates`

5. Build the Maven Project

With all NPM packages built and installed, it is now possible to build the Maven project.

- Using IntelliJ, this is automatically done by running one of the run configurations which include "Backend" in their name.
- It is also possible to build the JAR using Maven with the command: `mvn clean package`

6. Run the JAR File

Now it is possible to run the created JAR file as explained in Java Archive, or by running any of the IntelliJ run configurations.

IntelliJ Run configurations

This is a list of all the included IntelliJ run configuration which can be used to run different parts of the project.

Name	Description
Install	Install all Maven and NPM packages
Backend with Integrated Frontend	Build the Client, Email and Java code and run the project
Backend	Build the Email and Java code and run the project
Backend only	Build only the Java code and run the project
Frontend Only	Run only the frontend seperatly on http://localhost:5173/
Email Build Only	Build only the Email templates
Backend Test	Run only the backend JUnit tests

If a backend run is started without building the Client or Email the Client and Email are not updated which causes the backend to use the results of a previous Client and Email builds.

C.2.4 Docker

To deploy the project in a Docker container, it is possible to build your own image using if installed manually using the `Dockerfile` and the terminal command `docker build -t <image> <path`

Another way is to create a docker container with the `openjdk:23-jdk-slim` image and adding the JAR to it, however this requires additional docker configurations to execute the JAR when starting the container.

C.3 Project Structure

C.3.1 Folder Structure

Path	Description
.	Root directory containing the Maven project files, Gitlab CICD file and the Dockerfile
config	A folder automatically generated by the system on the first boot in which a ‘default-application.properties‘ will be placed with example values which need to be replaced before the system can be used properly, like the Database and email SMTP server configuration
.run	Contains IntelliJ run configurations to easily run different parts of the project.
src	Contains all the code of the project
src/main	Contains the functional code
src/main/client	Contains the code of the Client including its NPM package with dependencies
src/main/email	Contains the email templates and a NPM package to create HTML files which can be rendered by email clients
src/main/java	Contains all the Java source code. The only package in this folder is ‘nl.utwente.ps.inventory‘
src/main/resources	Contains all resources used by the Java code, including the settings and database schema
src/test	Contains all tests
src/test/java	Contains all JUnit tests to test the Java code in src/main/java
src/test/resources	Contains all additional settings which should be used instead of those in src/main/resources while running the JUnit tests.

C.3.2 Java Structure

The Java package of the code is `nl.utwente.ps.inventory` which contains a variety of other packages listed in the table below.

Package	Description
api	Contains all DTO's, REST Controllers and Services used by the system grouped by component.
config	Contains the Config Initializer which ensures a config file will be placed in /config
database	Contains all database entities and Enums as defined in the schema.sql
mappers	All mappers which can map database entities to DTOs
middleware	Contains all code used for authentication
repository	Contains all database repositories to get entities from the database using defined queries.
task	Contains all tasks which will be executed periodically, e.g. ScheduledReturnReminderTask for reminding users about their borrowed items.
utils	Utility package containing some useful code, e.g. UnixTime to store date as integers.
utils.ical	Contains everything that has to do with ICalendar (.ics) files which are sent by emails.

C.3.3 React Structure

The react component contains a variety of other packages listed in the table below.

Package	Description
components	Contains reusable components such as forms, modals, alerts and buttons that are used in the user interface.
data	Contains information regarding pages, such as the url, page title, subpages and permission level of a given page. This provides a well-organized configuration for the navigation system.
hooks	Includes reusable functions that handle api requests.
propTypes	Helps declare reusable proptypes to be used in components and pages.
scss	Contains the custom color scheme and styles of the website.
utils	Contains reusable helpers that help with how data is parsed, formatted, and displayed.

C.4 Configuration

For the system configuration, there are different config files of which some are located in the Jar and other in the `config` folder in the root directory. The files inside the JAR cannot be changed except when using the manual install, this is why the system creates a config folder outside the JAR containing configurations that should be provided by the user. As a result of Spring Boots configuration system, the configuration file in the `config` folder overwrites settings of the properties file inside the JAR, making it possible to change all settings of the system without touching the main configuration.

C.4.1 Main configuration

The main configuration is located in `src/main/resources/application.properties` which will be put inside the Jar and contains the necessary settings for the system to work properly.

After the server has started for the first time, the file `/config/application-default.properies` will be generated containing settings which need to be changed before the system can run properly, this includes the database and email configuration, but also the `web.address` which is needed for links in emails to point to the correct site.

It is possible to change values of the `src/main/resources/application.properties` configuration by settings the desired value inside `/config/application-default.properties`, which will then overwrite the setting in `src/main/resources/application.properties` while allowing you to always revert back to the previous setting if you remove the setting from the `/config/application-default.properties`.

C.4.2 Test configuration

The test package contains its own configuration file in `src/test/resources/application-test.properties` which will only be used when executing JUnit tests. This configuration replaces the database by a in-memory Hibernate H2 database which prevents the JUnit tests from running on the production database. Furthermore the `src/tst/resources/` contains its own database schema due to incompatibility issues between MariaDB and H2 due to `FULLTEXT` statements not being supported by H2.

C.5 Client Components

C.5.1 Form system

The form system is used to create state forms. These forms need to store the values even after re-renders. To make use of a StateForm you can define it the same way as a regular Form:

```
1    <StateForm action={...} asyncAction={...}>
```

```
        ...
    </StateForm>
```

Here the action and asyncAction are both optional parameters. When given a function is expected, which returns a boolean. The asyncAction function has the added option to be asynchronous in which case the response will be awaited. When returning a True value, the form data will be cleared, when returning a False value, the form data will be kept.

Withing the StateForm multiple input components can be used.

- CheckFormInput - Used for checkboxes, or other true/false values
- FormInput - A general form input, used for text or numbers
- FormSelect - A select dropdown with preset options which the user can choose from
- FormTextArea - A TextArea input, used for long amounts of text
- RadioFormInput - Used for radio inputs

Each of the components will use default classes, unless the `className` property is set. To give a default value, use the `defaultValue` property, make sure this matches the type expected by the input. Additionally, all inputs allow an optional required property to be set, when set the property is seen as a required property and the form can not be submitted without this input being set. If needed, the type of the input can also be changed for all inputs that support it (not supported by `FormTextArea`).

A name is required for all the input fields, this is the name under which the value of the input is stored. Re-using the same name for the same input fields, will sync the value between all the inputs with the same name. If no id is given, the id will be the same as the name.

The `FormInput` also allows additional options for the step, min and max values when the input is set to the type number. This is also the only input component where the type property is required. Additionally, it is also one of only input where an `onChange` property is added, together with `RadioFormInput`, where you can listen to a change in the input.

The `FormTextArea` also has a `rows` property, for how much rows the text area should have initially.

C.5.2 Modal system

To make use of the modal system, make sure the modal and any components interacting with the modal are within the `GlobalModalContext.Provider`. The provider should provider an instance of the `ModalContextProvider`.

```
    <GlobalModalContext.Provider value={new ModalContextProvider
        ()}>
        ...
    </GlobalModalContext.Provider>
```

In order to define a modal, the modal component should use the value returned by the `GlobalModalContext` and create a new context with an id and initial value. Whenever the context is created, a reference object is returned from which the value can be read. A code example of how to create a modal context is included below.

```
1    const modalContextProvider = useContext(GlobalModalContext);
2    const modalContext = modalContextProvider.createContext("
        Insert.Modal.Id", {
3        InitialData = null
4    });
```

The modal object should use the reference from the created context by using the `ref` property added by React.

```
1    <div className="modal␣fade" tabIndex="-1" aria-hidden="true"
        ref={modalContext.modalRef}>
2        ...
3    </div>
```

The option to add values is added in order for the modal to be easily reused and change the contents within the modal. To get the current information from the context use `modalContext.data`.

C.5.3 Bag system

In order to use the bag system, get a reference to the bag by adding |const bag = useContext(GlobalContext);—. The bag provides the following functions you can use for reading and updating the bag.

- bagCount() - Get the total count of items currently in the bag
- getCount(type, itemId) - Get the amount of items in a bag from a certain item ID. When the item type is 0 the itemId should be the id of the item. When the item type is 1 or 2, the itemId should be the id of the item collection.
- list() - Get all items currently in the bag
- addItems(collection, itemsOrCount) - See the "Adding items" section below.
- setItems(type, collectionId, newCount) - See the "Updating items" section below.
- removeItem(type, itemId) - Remove all of an item out of the bag. When the item type is 0 the itemId should be the id of the item. When the item type is 1 or 2, the itemId should be the id of the item collection.
- clear() - Remove all items from the bag.

Adding items

When adding items the full collection information should be given as collection. It at least expects the structure of:

```
{
    type: 0,
    id: 0,
    name: ""
}
```

The itemsOrCount should contain the amount of items that need to be added from a collection in the case of a bulk or infinite collection. However, it expects a list of item ids for individual collections, it then proceeds to add all these individual item ids to the bag, making it possible to borrow many items at once.

Updating items

The type (1 or 2) should still be given, individual items cannot be modified like this as their count can only be 0 or 1. When setting the new count to 0, it will remove the item from the bag using `removeItem`.

C.5.4 Alert system

The alert system does not require a `useContext` call, instead call `setAlert(level, contents)` to set a new alert. The contents are what is displayed in the popup, and the level determines the header of the popup. It can be any of the below:

- ”error” or ”danger” - A red colored header reading: ”Error”
- ”warning” - A yellow colored header reading: ”Warning”
- ”success” - A green colored header reading: ”Success”
- anything else - A blue colored header reading: ”Info”

C.5.5 Sessions

The session system can be used by gathering a reference similar to the bag reference, |const session = useContext(GlobalContext);—. The following functions are available on the session object:

- hasSession() - Returns a boolean, whether the user has a valid session.
- isEmployee() - Returns a boolean, whether the user is an employee (True for employees and administrators, False for guests)
- isAdmin() - Returns a boolean, whether the user is an administrator (True for administrators, False for guests and employees)
- createSession(account, token, expiration, role) - Store new session information. See the ”Create session” section.
- destroySession() - Remove the session information from the system.

Create session

The account variable contains the user information. This account information is currently not used in the front-end, and not possible to get from the session object.

The token should be the token that needs to be added to the header of requests, in order for the back-end to authorize the request.

The expiration should be the end date of the session. The session will be automatically invalided (but not removed) once this date passes.

The role should be the role of the user that just logged in, "guest", "employee", or "admin".

C.6 Server Components

C.6.1 Database

For the database a custom `src/main/resources/schema.sql` file has been created to create or database. This schema should be modified or extended if a component which needs to be stored in the database is created.

All tables inside the database get their own entity in the `nl.utwente.ps.inventory.database` package where all datatypes match those in the database and foreign keys map to the Class containing the Entity a Foreign key is linked to including the type of mapping. The entities in the code are mapped to the database using Hibernate allowing for the creation of repositories in `nl.utwente.ps.inventory.repository` which can create, retrieve, update and delete data from the database.

Below is the database UML diagram containing a visual representation of the schema.sql.

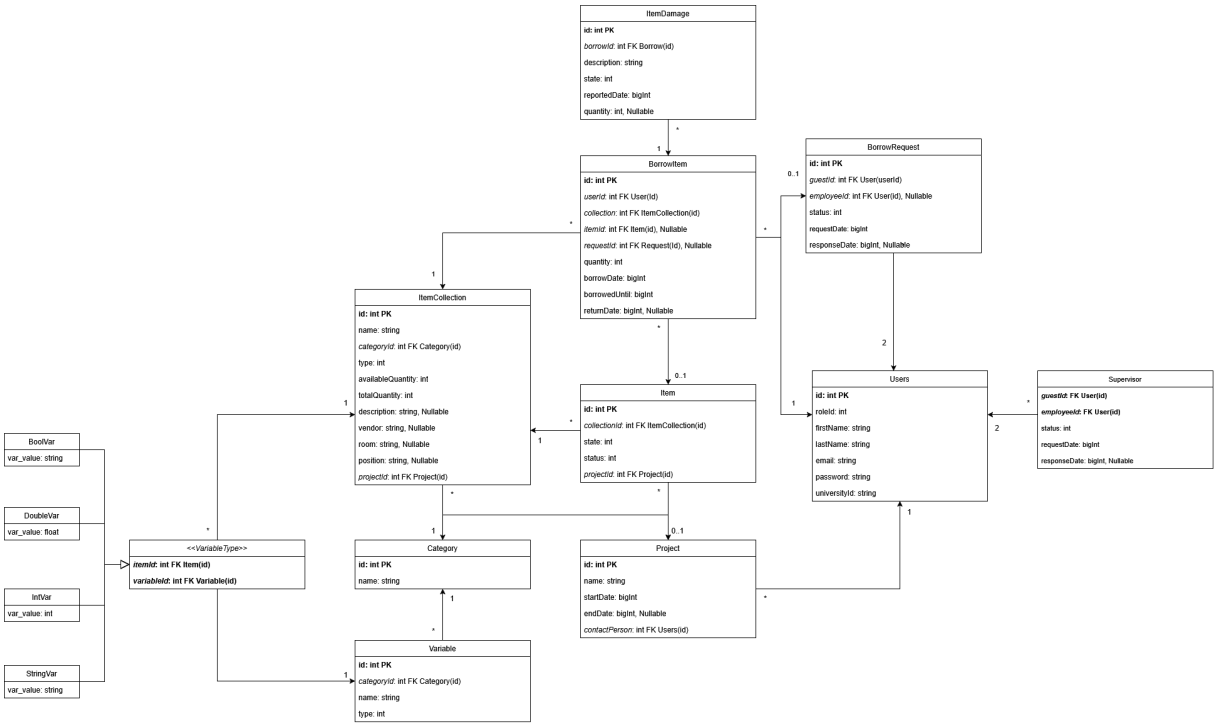


Figure C.1: Database Schema

Entity

All entities are annotated with `@Entity` and
`@Table(name="<TableName>")` to ensure that Hibernate uses the correct table. All entities in the system have a id property which is the Primary Key of the table and is annotated with `@Id`,
`@GeneratedValue(strategy=GenerationType.IDENTITY)` and
`@Column(name="id",nullable=false)` to map the value to the correct database column. All other properties are only mapped with `@Column` with the respective column names and if the value can be nullable. The foreign keys are joined columns annotated by their relation (ManyToOne, OneToOne) and `@JoinColumn(name="fk_id_column")` to ensure the correct mapping to the related Entity.

Some values which are Integers inside the database get an Enum as its type in their Entity instead. This has been done since these integer values only support a number of options which would be coded eitherway and creating Enums made this easier. All Enums contain a value which represents the integer value to be stored in the database or given in a front-end request body. To accomplish this, the `nl.utwente.ps.inventory.database.BaseValueEnum` interface has been created which is implemented by all the entity enums which can be seen in the table below including their values.
The correct mapping from an Entity column to the Enum the
`@Enumerated(EnumType.ORDINAL)` annotation needs to be used, since the value of all values correspond to their ordinal value.

Enum	Values
ItemCollectionType	UNIQUE(0), BULK(1), INFINITE(2);
ItemCondition	GOOD(0), DAMAGED(1), BROKEN(2);
ItemStatus	AVAILABLE(0), BORROWED(1);
ResponseStatus	PENDING(0), APPROVED(1), REJECTED(2);
Role	GUEST(0), EMPLOYEE(1), ADMIN(2);
VariableType	BOOL(0), INT(1), DOUBLE(2), STRING(3);

Repository

Repositories contain all queries to manipulate the database with the above mentioned entities. Most queries are defined by name only, so e.g. the function `Category getByName(String name);` returns the `Category` with the given `name`, which is equivalent to a SQL query like `SELECT c FROM Category c WHERE c.name = :name`. Some more complex queries cannot be defined by their function name only, e.g. if you want to get a User Entity from a BorrowedItem. An example for such a function definition is given below.

```
1    @Query(value = "select␣distinct␣i.user␣from␣BorrowItem␣i␣
        where␣i.returnDate␣is␣null")
2    List<User> findAllUsersWithBorrowedItems();
```

Mapper

For mapping the MapStruct library is used. Mapping is used to transform a database Entity into a DTO which can be used by all API components since these cannot directly modify the data stored in the database. For each entity there exists a unique mapper interface or abstract class in `nl.utwente.ps.inventory.mapper`, of which most only contain a simple `toDTO` and `toEntity` of which some also have a separate `@Mapping(source, target)` mapping to ensure some objects are mapped to the correct place, e.g. a `guestId` field in a DTO needs to be mapped to the `guest.id` field in the Entity.

Some mappers contain more functions if there exist more DTO's for a specific component, e.g. the Collection component contains four distinct DTO's which all need to be mapped correctly to the ItemCollection entity.

Abstract class mappers are not much different from their interface variants other than they contain an implemented function which validates that some values are set, e.g. the `UserMapper.toEntity(UserSignupDTO)` function sets the default role of a newly created user to that of a `Role.GUEST`.

C.6.2 API

The `nl.utwente.ps.inventory.api` is seperated in all the different components to make the structure more clear which makes it easier to locate classes. Each seperate component

contains at least one DTO, Service and RestController. The only exceptions to this is the `exception` package, which does not contain any of these classes and is explained more detailed in Error Handling.

DTO

A Data Transfer Object (DTO) is used as object which transfers data from client to server using JSON requests, but it is also used as the default object in RestControllers and Services to ensure database safety since DTO's cannot be inserted directly into the database. DTOs contain all data that is needed for a specific request or response used for the communication between client and server, the RestControllers accept a DTO as request body use it as response object.

DTOs do not have specific annotations except Lombok annotations for cleaner classes, and some DTOs make use of Jakarta annotations which validate the RequestBody that is received by the server.

All DTO classes end with DTO in their name.

RestController

The RestControllers are the part of the system the client can reach using HTTP requests. These classes are always annotated by `@RestController` and `@RequestMapping("/api/...")` where `/api/...` represents the url path, e.g. for the category API this is `/api/category`. All functions in the controllers are annotated with a HTTP Method mapping, the possiblities are `@GetMapping("")`, `@PostMapping("")`, `@PutMapping("")` and `@DeleteMapping("")`. Each of these mappings represents one of the HTTP methods, and the string inside these mappings are paths to the respective mapping which are added after the `RequestMapping` as defined for the class.

A method mapping can contain a normal path e.g. `"/item"`, or it can contain path parameters like `"/{id}"` where id represents a id, e.g. `"/api/category/3"` is a valid mapping for `@GetMapping("/{id}")` inside the `CategoryController`, where 3 would be used as the id.

Each controller mapping almost directly calls and returns the result of its Service class, since the service handles the validation and execution of any required code like database queries. In some methods an email is also sent, this is not handled by the service since it is only request dependent an it calls the separate `EmailService` for this. More about emails can be read in section C.6.4. Other functions of the controller methods can be to set some initial values when a null value is given.

Service

All service classes handle a part of the business logic of the system seperated by the different components. First of all the services should validate all given DTO objects, since this is mostly provided by a client which can be malicious or incomplete data the server should reject. When a service received invalid date, e.g. a object with the given ID does not exist or the name is already in use an custom exception should be thrown

as explained in section C.6.5. If the given DTO object is valid the service can continue with its task, e.g. updating or retrieving an entity, and it should return a DTO when completed if required. A service method should only return DTOs and never Entities since entities should never be sent to the client.

C.6.3 Variables

For this project a custom variable system has been implemented which allows for storing variables with different types of values. The support values are Boolean, Integer, Double and String which are all store in seperate database tables, BoolVar, IntVar, DoubleVar and StringVar respectively. Using different tables for each value type made the variables modular and easy to create new types of variables if needed. These values will be referenced to as TypeVars in the remaining part of this section.

All TypeVars have a foreign key linked to the ItemCollection they are a value of and the Variable which holds the name, type and link to the category it is part of. Each item collection has its own values for the variables inside the category the collection is part of.

Implementation

Since typevars are are similar except their value type an abstract Entity class `TypeVar<T>` has been created where `T` is the value type of the typevar. For each type var a separate class extending `TypeVar<T>` has been implemented, e.g. `IntVar extends TypeVar<Integer>`.

For the Variable Entity the types are defined by the `VariableType` enum which contains the integer type values (See section C.6.1), an constructor for the specific TypeVar and the class of the `T` value. The constructor is used to create the correct object and the value type is used to validate any value to be set into a TypeVar such that the type matches.

The TypeVar DTOs have been implemented similarly to the entities, however there are some annotations added to be able to parse the separate type vars using Jakarta and a custom `TypeVarTypeIdResolve` has been implemented in the `nl.utwente.ps.inventory.api.typevar` package. The following annotation has been added to the `TypeVarDTO<T>` class:

```
@JsonTypeInfo(
        use = JsonTypeInfo.Id.CUSTOM,
        include = JsonTypeInfo.As.EXISTING_PROPERTY,
        property = "type"
)
@JsonTypeIdResolver(TypeVarTypeIdResolver.class)
@JsonSubTypes({
        @JsonSubTypes.Type(value = BoolVarDTO.class, name = "0"),
        @JsonSubTypes.Type(value = IntVarDTO.class, name = "1"),
        @JsonSubTypes.Type(value = DoubleVarDTO.class, name = "2"
            ),
        @JsonSubTypes.Type(value = StringVarDTO.class, name = "3"
            )
})
```

The defined `JsonSubTypes` represent the different type var classes as value with their type ID as name. In the `TypeVarTypeIdResolver` class the different types are implemented as follows:

```
    @Override
    public JavaType typeFromId(DatabindContext context, String id
       ) {
        TypeFactory typeFactory = context.getTypeFactory();
        return switch (id) {
            case "0" ->
                    typeFactory.constructSpecializedType(
                       typeFactory.constructType(TypeVarDTO.class)
                       , BoolVarDTO.class);
            case "1" ->
                    typeFactory.constructSpecializedType(
                       typeFactory.constructType(TypeVarDTO.class)
                       , IntVarDTO.class);
            case "2" ->
                    typeFactory.constructSpecializedType(
                       typeFactory.constructType(TypeVarDTO.class)
                       , DoubleVarDTO.class);
            case "3" ->
                    typeFactory.constructSpecializedType(
                       typeFactory.constructType(TypeVarDTO.class)
                       , StringVarDTO.class);
            default -> throw new IllegalArgumentException("
               Unknown␣type␣ID:␣" + id);
        };
    }
```

These things need to be extended when a new type var is implemented.

C.6.4 Email

The different types of emails that can be sent are defined in the `EmailService` interface in the `nl.utwente.ps.inventory.api.email` package. The interface has two implementations, the `EmailServiceImpl}` which is inside `src/main/java` and the `EmailServiceTestImpl` inside `src/test/java`. In `EmailServiceImpl` the emails are created and sent, but this is not a desired functionality for tests, so the `EmailServiceTestImpl` is an empty implementation without any functionality which is only used by the test classes. The `EmailService` can be autowired in RestControllers to sent the desired emails, e.g. when a user borrows items:

```
    @PostMapping("/")
    public List<BorrowItemDTO> addBorrowItems(@RequestBody List<
       BorrowItemDTO> body) {
        // ...
        try {
            emailService.sendBorrowed(user,
               successfullyAddedItems);
        } catch (Exception e) {
            logger.error(e.getMessage(), e);
```

```
        }
        // ...
    }
```

It is possible there went something wrong while sending the email, in that case the exception is catched and logged to the console to not fail the request. Due to latency issues while sending emails, all email service methods have been made `@Async` such that a response is returned directly without waiting on the email.

Templates

The content of the emails is created using predefined templates in the `src/main/email/templates` folder. These templates are HTML files containing a template which will render properly in an email client, however some email clients have a better HTML implementation than others (e.g. a web based client like Gmail supports HTML based emails better then an app like Outlook).

The style sheet used for the templates is located in the `style.css` which should be inlined into the HTML templates, which is done automatically when building the email templates as explained in section C.2.3.

The templates also make use of Thymeleaf which allows for adding variables into the HTML code allowing for customized messages and even tables, like the tables for borrowed and returned items. Variables can be set in a `TemplateEmailDTO` which will make sure that these variables are processed when processing the Thymeleaf template using the `EmailServiceImpl.sendTemplateEmail` function. This function will also sent the email directly to the user set in the `TemplateEmailDTO` and it allows to add email attachments like ICalendar files.

ICalendar

ICalendar or .ics files are files which contain standardized calendar information which can be read by almost any calendar application which makes it a suitable format to sent borrow return date reminders to users. The ICalendar classes are a custom implementation located in `nl.utwente.ps.inventory.utils.ical` and is able to generate RFC 5545 compliant .ics file contents containing a calendar with multiple events.

Using the `ICalGenerator` it is possible to easily create an ICalendar containing all return dates of borrowed items. This functionality is used in the `EmailService` to add EmailAttachments with a ICalender file for when users borrow items, or a return reminder is sent.

Due to some inconsistent behaviour of ICalender implementations in various email clients it is possible that updated borrow item ics files do not overwrite a previoulsy added ics files even when the Event ID is similar which can result in duplicated calendar notifications.

C.6.5 Error Handling

It is possible that a user makes an invalid requests, or some part of the system is not working as expected. To ensure that the system does not crash all encountered exceptions caused by user requests are catched and returned as a response to the user by the

`api.exception.GlobalExceptionHandler`. If the catched exception is caused by a user error, an HTTP 4xx status code will be responded with the cause of the error such that the user can correct their mistake. If the excepiton is caused by the server, a 5xx status code will be responded with the message that something went wrong on the server. We chose to hide the real 5xx error messages from the user since these can contain critical system information which the user does not need access to.

All exceptions that can be thrown by the system for user errors are all located in `api.exception.*` as claasses which extend `RuntimeEsception`. To ensure that the proper response status codes and messages are returned by the `GlobalExceptionHandler` is in all classes a `@ResponseStatus(value=HttpStatus.*)` annotation and a `String` message or a constructor which takes a `String`. Below is the CategoryNotFound exception given which is thrown when a category with a requested ID does not exist, his exceptions returns a 404 Not Found response to the user including the "Category ID does not exist" message.

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class CategoryNotFoundException extends RuntimeException {
    public CategoryNotFoundException(Integer id) {
        super("Category ’" + id + "’ does not exist");
    }
}
```

The last class located in the `api.exception` package is the `ApiErrorHandler` which catches all exceptions thrown by the `Servlet` layer. The `Servlet` layer is part of the springboot layers, however this layer is called before a call is made to our own classes.

C.6.6 Scheduler Task

A scheduler task is a task which is executed on a predefined timestamp. The only task implemented is the `ScheduledReturnReminderTask` which is configured by a cron timestamp defined in the `config/application-default.properties` file to execute every day at 9.00 AM to send return reminders for items which are overdue, due today or due tomorrow. It is also possible to send once a week a complete overview including items due this week and next week, by default this is on monday and configurable in the config.

C.6.7 Security

Middleware

The security of the system is handled by the middleware and user login. After a user logs in they receive a token which they need to put in their header, e.g.
`Authorization: Bearer <Token>`. If the client tries to access any path the middleware checks if the path can be accessed always
(`JwtAuthenticationFilter.authorizationFreePaths` and non-api paths), only without authorization header
(`JwtAuthenticationFilter.preAuthorizationPaths`) or only with an authorization header (all other `/api/*` paths). If the client tries to access an invalid path or the authorization header is invalid a 401 Unauthorized response will be returned to the client.

If the client has a valid authorization header the session of the client will be stored such that it can be used by the `UserService` to retrieve the user session.

Permissions

Not all API endpoints are accessible for all users, some are only available for Guests while others only for Admins. To manage this an annotation has been created which can be added to any method in a RestController which allows to manage access, the `@PreAuthorize("hasAuthority('ROLE')")` annotation. If the annotation is not added to a method, the method is accessible by all logged in users no matter their role. These permissions are defined in`nl.utwente.ps.inventory.api.user.CustomUserDetails`

Role	Permission
Guest	Only Guests can access this method
Employee	Only Employees AND Admins can access this method
Admin	Only Admins can access this method

C.7 Testing

For most API components a JUnit test has been created. These tests mostly test the services, since these classes contain the most critical code which makes changes in the database. Some tests also contain HTTP request tests that validate the responses of HTTP requests.

C.7.1 Annotations

All test classes are annotated with `@SpringBootTest` to register the class as a test and to ensure the Spring Boot test profile is used all test classes are annotated with `@ActiveProfiles("test")`. This ensures the test classes use the `src/test/application-test.properties` file containing a in-memory H2 database, otherwise the tests will alter data inside the database defined in `config/application-default.properties`.

C.7.2 Testing Spring Boot Components

The Spring Boot tests make use of the `@Autowired` annotation to use Spring Boot components like services and repositories which are needed to test these components.

C.7.3 Testing HTTP Requests

For testing HTTP requests a `MockMvc` needs to be created which can be used to test requests which cannot be autowired. Since most requests require a Authentication token in their header it is also recommended to create a logged in user using the `ApiUtils` which contains a `token` that can be put inside a header. An example of an implementation which creates a `MockMvc` and `userSession` for each test and has a `testRequest` test which checks if the request to `/test` returns a 2xx statuscode can be seen below:

```java
@SpringBootTest
@ActiveProfiles("test")
class HttpRequestTest {
    private MockMvc mockMvc;
    private UserService.LoginResponse userSession;

    @BeforeEach
    void init() {
        // ensure the database is empty
        apiUtils.deleteAll();
        mockMvc = MockMvcBuilders.webAppContextSetup(
            webApplicationContext).build();
        userSession = apiUtils.getLoggedInUser(Role.ADMIN);
    }

    private HttpHeaders getHeaders() {
        HttpHeaders headers = new HttpHeaders();
        headers.add(HttpHeaders.AUTHORIZATION, "Bearer " +
            userSession.token());
        return headers;
    }

    @Test
    void testRequest() {
        mockMvc.perform(get("/test").headers(getHeaders()))
                .andExpect(status().isOk())
    }
}
```

Note: The first user always needs to be an Admin, if you want to create an Guest or Employee user two users need to be created, a Admin and a user with the required role.

C.7.4 Api Utils

The class `test/java/nl.utwente.ps.inventory.utils.ApiUtils` has been created to make creating tests easier. This class contains multiple helper functions to create objects like users, categories and categories which also stores them in the Database allowing test classes to create objects using the services they need to test without having to create new objects they need for creating such objects, e.g. multiple Users are needed to create a Supervisor and by calling `apiUtils.createUser(Role)` twice two users are created in the system which can immediately be used in tests.

Below is the implementation of `ApiUtils.createCategory()` given as a reference. This function generates a random UUID v4 which will be used as the category name to ensure unique names, then a BaseCategoryDTO is created without DTO which is then used to create a category using the `CategoryService`.

```java
    public BaseCategoryDTO createCategory() {
        String name = UUID.randomUUID().toString();
        BaseCategoryDTO category = new BaseCategoryDTO(null, name
            );
```

```
4
5           return categoryService.createCategory(category);
6       }
```

The `nl.utwente.ps.inventory.utils.ApiUtils` is only available inside tests and is not usable in the main system.

Functions inside `ApiUtils` should only be used outside of test classes where the respecitive component is tested, so the `createCategory` function should not be used inside the `CategoryControllerTest` since the `ApiUtils` assumes the `CategoryService` works correctly.

C.8 Code Style and Conventions

C.8.1 Client

The client follows the React code conventions and makes use of ES Lint for clean code.

C.8.2 Server

The server follows the Java code conventions, uses Lombok annotations (e.g. `@Getter` and `@Setter`) to reduce boilerplate code to make classes look cleaner.

For good practice only the Service layers handle business logic such that Rest Controllers, DTOs and Repositories do not have to validate any code for the system to work properly. The DTOs make use of Jakarta annotations like `@Nonnull` to ensure request bodies provided by the client are valid and can be used properly by the server.

C.9 Further extensions

C.9.1 Orders

Orders are a way for a system admin to add items which are being ordered to the system to let users know that a certain item is being ordered when it is not yet in the system.

Orders were part of the should haves of the project and were only implemented on the server and database side of the system. Most of the controller and service functions have also already been implemented in the `nl.utwente.ps.inventory.api.order` package, but due to time constrains it was not feasible to fully implement this functionality of the system.

C.9.2 Static ICal URLs

By using static ICal URLs instead of providing a new ics file with every email for the borrowed items, it would be possible for some email clients to properly update calendars which can resolve the issue where duplicated borrow reminders are added to a calendar.

C.10 Used dependencies

The table below contains an overview of all used dependencies of the client, email and server packages including a short description and an URL to the project information.

Name	**Description**	**URL**
Spring Boot	Java framework to create stand-alone applications	https://spring.io/projects/spring-boot
Lombok	Java annotations to simplify code	https://projectlombok.org/
MapStruct	Simplified Java bean mapping	https://mapstruct.org/
Thymeleaf	Template engine for Java used by the email templates	https://www.thymeleaf.org/
Bootstrap	Frontend toolkit to customize the user interface	https://getbootstrap.com/
React	Frontend framework for building the user interface	https://react.dev/
Juice	Email template CSS inliner tool	https://github.com/Automattic/juice

Bibliography

Adams, A., & Sasse, M. A. (1999). Users are not the enemy. *Commun. ACM*, *42*(12), 40–46. https://doi.org/10.1145/322796.322806

Böck, H. (2011). Java persistence api. In *The definitive guide to netbeans™ platform 7* (pp. 315–320). Apress. https://doi.org/10.1007/978-1-4302-4102-7_28

Cadran. (2025). *Oracle netsuite erp services - cadran* [Accessed: 2025-03-05]. https://www.cadran.nl/services/oracle-netsuite-erp/

Ebad, S. A. (2022). Exploring how to apply secure software design principles. *IEEE Access*, *10*, 128983–128993. https://doi.org/10.1109/ACCESS.2022.3227434

Gamma, T., & Gerasimenko, N. (2024). *React security labs* [Doctoral dissertation, OST Ostschweizer Fachhochschule].

ShipHero. (2025). *Shiphero - ecommerce fulfillment & warehouse management* [Accessed: 2025-03-05]. https://shiphero.com/