FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Lukáš Holub

Design and implementation of an item positioning management system

Department of Software Engineering

Supervisor of the bachelor thesis: Ing. Pavel Koupil, Ph.D.

Study programme: Programming and software development

Prague 2026

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to express my sincere gratitude to my supervisor, Pavel Koupil, for his continuous guidance, valuable advice, and support throughout the preparation of this thesis. I am also deeply grateful to my family for their constant care, patience, and encouragement, which made the completion of this work possible.

Title: Design and implementation of an item positioning management system

Author: Lukáš Holub

Department: Department of Software Engineering

Supervisor: Ing. Pavel Koupil, Ph.D., Department of Software Engineering

Abstract: Keeping track of the location of physical items is a common problem in both personal and small-scale business environments. Existing inventory management solutions often focus on quantity tracking rather than spatial organization and typically require significant domain knowledge, leading users to rely on informal methods such as spreadsheets that do not scale well and are prone to inconsistency and errors. This thesis focuses on the development of a visual item location management system that supports spatially accurate tracking of physical items. The system allows users to model real-world spaces using customizable floor plans and furniture representations, providing visual context for item placement and organization, while supporting intuitive item placement, movement, and real-time location lookup. Compared to existing solutions, the system emphasizes spatial accuracy, usability, and extensibility, making it particularly suitable for personal use and small organizations without professional inventory infrastructure.

Keywords: inventory system, interactive editors, spatial organization

Název práce: Návrh a implementace systému pro správu umístění položek

Autor: Lukáš Holub

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Ing. Pavel Koupil, Ph.D., Katedra softwarového inženýrství

Abstrakt: Udržování přehledu o poloze fyzických předmětů je běžným problémem jak v osobním, tak v menším podnikatelském prostředí. Stávající řešení pro správu zásob se často zaměřují spíše na sledování množství než na prostorovou organizaci a obvykle vyžadují značné odborné znalosti dané oblasti, což vede uživatele k tomu, že se spoléhají na neformální metody, jako jsou tabulkové procesory, které nejsou dobře škálovatelné a jsou náchylné k nekonzistencím a chybám. Tato práce se zaměřuje na vývoj vizuálního systému pro správu polohy položek, který podporuje prostorově přesné sledování fyzických předmětů. Systém umožňuje uživatelům modelovat reálné prostory pomocí přizpůsobitelných půdorysů a reprezentací nábytku, které poskytují vizuální kontext pro umístění a organizaci položek a zároveň podporují intuitivní umístění položek, jejich přesun a okamžité vyhledávání polohy. Ve srovnání se stávajícími řešeními klade systém důraz na prostorovou přesnost, použitelnost a rozšiřitelnost, díky čemuž je vhodný zejména pro osobní použití a malé organizace bez profesionální infrastruktury pro správu zásob.

Klíčová slova: inventarizační systém, interaktivní editory, prostorová organizace

Contents

Introduction **8**

1 Related Works **10**
1.1 Sortly . . . 10
1.2 CyberStockroom . . . 11
1.3 Ikea . . . 12

2 Analysis **14**
2.1 Functional Requirements . . . 14
2.2 Non-functional Requirements . . . 15
2.3 User Roles . . . 16
2.4 Use Cases . . . 18
2.4.1 View the inventory list . . . 19
2.4.2 Track the location of a specific item . . . 20
2.4.3 Add, edit, or remove a shape . . . 21
2.4.4 Add a furniture . . . 25
2.4.5 Add a floor . . . 27

3 Design Documentation **29**
3.1 C4 Model . . . 29
3.1.1 System Context Diagram . . . 29
3.1.2 Container Diagram . . . 30
3.1.3 Component Diagrams . . . 32
3.2 Domain Model . . . 35
3.3 Technologies . . . 41
3.3.1 Frontend Technologies . . . 41
3.3.2 Backend Technologies . . . 43
3.3.3 Database Technologies . . . 44
3.3.4 Build and Deployment Tools . . . 44
3.4 Mockups . . . 45
3.4.1 Item Location Mockup . . . 46
3.4.2 Furniture Editor Mockup . . . 47
3.4.3 Item List Mockup . . . 48

4 Developer Documentation **50**
4.1 Implementation Summary . . . 50
4.2 Algorithms and Data Structures . . . 51
4.2.1 Editor Scenes . . . 51
4.2.2 View Scenes . . . 52
4.2.3 Item Location Scene . . . 56
4.2.4 Undo/Redo . . . 56
4.2.5 Shape Types . . . 57
4.2.6 Adding Shapes to the Scenes . . . 58
4.2.7 Adding Label to Shapes . . . 59
4.2.8 Save/Load Shapes . . . 60

4.2.9 Moving Shapes . . . 60
4.2.10 Rotating Shapes . . . 60
4.2.11 Resizing Shapes . . . 61
4.2.12 Selecting Shapes . . . 61
4.2.13 Infinite Editor World . . . 62
4.3 User Interface . . . 65
4.3.1 Element Lists in Editors . . . 65
4.3.2 Element Modals in Editors . . . 66
4.3.3 Save Buttons for Editors . . . 67
4.3.4 Undo/Redo for Scenes . . . 67
4.3.5 List Pages . . . 67
4.4 Backend . . . 68
4.5 API . . . 72
4.5.1 Undo/Redo . . . 72
4.5.2 Add New Shape Types . . . 72
4.5.3 Add New Shape Manipulation Manager . . . 72
4.5.4 Add New Camera Manager . . . 73
4.5.5 Add New Shape Creation Manager . . . 73
4.5.6 Register New Shape . . . 73
4.5.7 Adding a New Scene . . . 74
4.5.8 Adding New Element Modals . . . 74
4.5.9 Defining Modal Field Schema . . . 74
4.5.10 Adding New Backend Repository . . . 75
4.5.11 Adding New REST Controllers . . . 77
4.5.12 Adding New Backend Entities . . . 77
4.5.13 Creating New Management Service . . . 77
4.6 Implementation Problems . . . 77
4.6.1 Undo/Redo . . . 77
4.6.2 Getting Correct Position And Size . . . 77
4.6.3 Fixed World . . . 78
4.6.4 Using Native Browser Scrollbars For Moving Camera . . . 78
4.6.5 String Identifiers In The Shape Manager . . . 78
4.6.6 Scene Not Resetting Upon Restart . . . 79
4.6.7 Dragover Event Not Triggered In Phaser When Dragged From Outside Phaser . . . 79
4.6.8 Undo/Redo For Item Movement . . . 80
4.6.9 Item List Page Action Button Dropdown Render Problem 80
4.6.10 Issues With Composite Keys In Spring . . . 80
4.6.11 Saving Data As JSON In Spring . . . 81
4.6.12 Adding New Path To Gateway . . . 81
4.7 Testing . . . 82

5 User Documentation **83**
5.1 Administrator Handbook . . . 83
5.1.1 Hardware Requirements . . . 83
5.1.2 Software Requirements . . . 83
5.1.3 Deployment . . . 84
5.2 User Handbook . . . 86

5.2.1 Home Page 86
5.2.2 Shape Editor 87
5.2.3 Furniture Editor 88
5.2.4 Floor Editor 90
5.2.5 Item List 91
5.2.6 Shape List 92
5.2.7 Furniture List 94
5.2.8 Floor List 94
5.2.9 Views 94
5.2.10 Item Location 94

Conclusion **96**

Bibliography **99**

List of Figures **100**

List of Tables **101**

List of Abbreviations **102**

Introduction

Keeping track of the location of physical items is a persistent challenge in both personal and business contexts. Even in small environments, users often waste time searching for infrequently used objects, leading to frustration and decreased efficiency. Traditional inventory systems, e.g. Sortly[1], usually do not provide user-friendly location information, particularly when items are scattered across different rooms or environments. In addition, in environments with a small number of people, informal methods are usually used to keep track of inventory, such as spreadsheets [1, 2]. With a growing number of items, it then becomes exponentially difficult to keep the data consistent with no errors.

For personal and small business contexts, an app that allows users to visually map and manage item locations without any experience and without special mental overhead would be an improvement compared to existing systems. That is why the goal of this thesis is to implement an application where the user can recreate his floor plans, and furniture, allowing to create a one-to-one copy of a real space requiring item management. It should allow even a new person without knowledge of the specific storage solution to locate easily existing items or change their location. The following are the specific goals for this application.

- Provide Real-Time Location lookup: Enable users to query and locate items quickly within a custom space definition.
- Support Custom Floor and Furniture Design – Allow users to create their own floor layouts and furniture structures to match their actual spaces.
- Offer Reusable Libraries – Build a community-driven library of shapes, floors, and furniture for reuse and inspiration.
- Allow Item Management – Enable adding, removing, and moving items between locations with intuitive visual controls.
- Minimize Mental Load – The system should be able to be used without a high mental load. It should be easy to use even if the user is new or has not used the system in a while.

Outline This thesis is structured as follows. Chapter 1 reviews related work and existing solutions relevant to item and inventory management systems and layout design. In particular, it examines current tools such as Sortly, CyberStockroom, and the IKEA digital planning tools, among others, highlighting their features and limitations. Chapter 2 presents the system analysis, defining both functional and non-functional requirements of the proposed application. This chapter also introduces the identified user roles and describes the primary use cases that guide the system design. Chapter 3 documents the system design. It begins with an architectural overview using the C4 model[3], including system context, container, and component diagrams. The chapter then introduces the domain model, explains key design decisions, and describes the selected technologies for

[1] https://www.sortly.com/

frontend, backend, database, and deployment. Finally, user interface mockups are presented to illustrate the expected look of the pages. Chapter 4 provides the developer documentation and focuses on the implementation of the system. It discusses deviations from the initial design where applicable and describes the frontend and backend implementations in detail. Finally, it describes the testing approach used to validate the system. Chapter 5 contains the user documentation and serves as a practical guide for deploying and using the application. It includes an administrator handbook covering system requirements and deployment steps, followed by a comprehensive user handbook that explains the functionality of the editors, lists, views, and item location features from an end-user perspective. Finally, the conclusion summarizes the work carried out in this project, evaluates the achieved results in relation to the defined objectives, and outlines possible directions for future work and further development.

1 Related Works

This chapter compares my proposed application with several other existing tools that partially overlap with the functionality of the proposed application.

The following list provides an overview of some of the related works:

- **Sortly**[1] – an inventory management system focused on item organization and metadata.
- **CyberStockroom**[2] – an inventory management system that tries to incorporate visual mapping of the warehouse.
- **IKEA Planners**[3] – a collection of different interior planning tools relevant to comparing editor functionality.
- **RoomSketcher**[4] – a user-friendly floor planning tool used for interior design and visualization.
- **Archicad**[5] – a professional computer-aided design (CAD)/building information modeling (BIM) tool for architectural modeling and construction planning.
- **Odoo Inventory**[6] – an inventory management system with a wide range of features such as automation and optimization of inventory management that works as a module for the Odoo ecosystem.
- **SketchUp**[7] – a general-purpose three-dimensional (3D) modeling tool which is often used for indoor layout design.

The following section provides detailed comparisons with selected applications from the list above.

1.1 Sortly

Sortly[8] is an inventory management system primarily oriented towards tracking items rather than their precise physical location. Although Sortly does allow users to organize items into nested folders representing storage areas, and additionally take photos of the item to indicate the location. These methods are not ideal for accurate and maintainable location tracking. Photographs would have to be updated on every movement and folder labels would have to be highly descriptive to be useful. A generic label such as `Storage Shelf` is insufficient in an environment where multiple identical shelves exist. Even when more detailed naming conventions are used (e.g. `Storage Shelf A1`), only individuals that are familiar with the organization's internal system will understand the item's true location.

[1] https://www.sortly.com/
[2] https://www.cyberstockroom.com/
[3] https://www.ikea.com/cz/en/planners/
[4] https://www.roomsketcher.com/
[5] https://www.graphisoft.com/plans-and-products/archicad/
[6] https://www.odoo.com/app/inventory
[7] https://sketchup.trimble.com/en
[8] https://www.sortly.com/

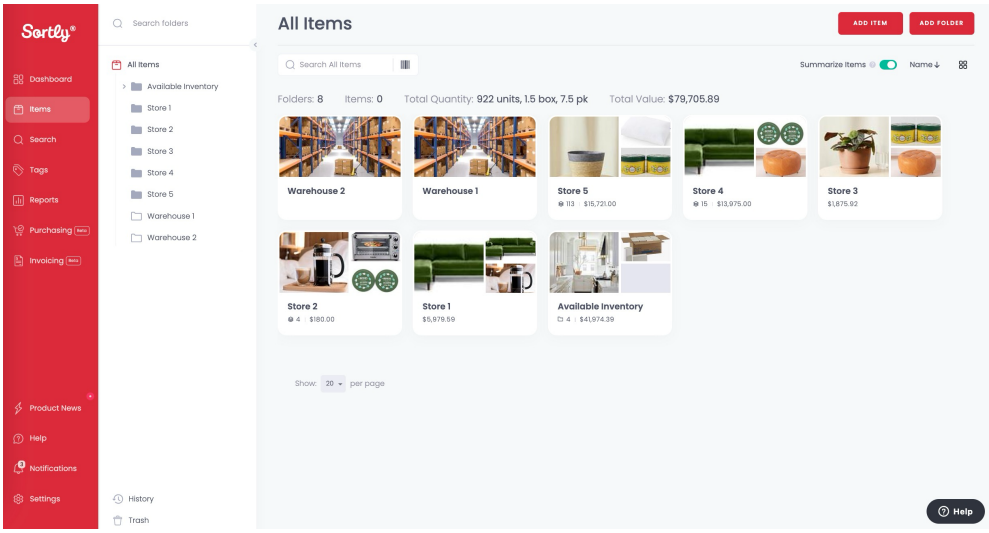


Figure 1.1 Sortly Items Page

In contrast, my application highlights the exact shelf and zone in a spatially accurate representation of the environment, enabling any user to intuitively locate any items without prior knowledge of the naming conventions. Although, within each zone, my system will work similarly to Sortly, by presenting items in a hierarchical structure where user will have to use descriptive names to identify specific boxes, or items. The reason for that is, that while it would be possible to implement editors on the item level, it would introduce some usability challenges. The users would have to manually update the location of each item while moving them between zones, making batch operations almost impossible. Additionally, deeply nested item structures would be very cumbersome to navigate since item structures can technically have infinite depth. For this reason, items within a zone are displayed as a tree, leaving it to the user to identify, for example, which specific box contains the desired item, with the system helping by highlighting the item path to the item.

Although Sortly is not optimized for detailed location tracking, it offers a robust set of features for item management. These features include, among other things: barcode scanning advanced metadata handling, and low-stock alerts. These capabilities are currently outside the scope of the current project, which focuses primarily on precise spatial location. As a consequence, item management features in this project will be a bit more limited, with mainly support for moving items between locations. In a future version of the application, many of Sortly's features would be a valuable addition to the application.

1.2 CyberStockroom

CyberStockroom[9] is an inventory management system focused on visual organization. Although its interface makes locating items a bit easier compared to Sortly, it still suffers from the same core limitation. The system also heavily relies on user-defined descriptive labels to indicate physical locations.

[9] https://www.cyberstockroom.com/

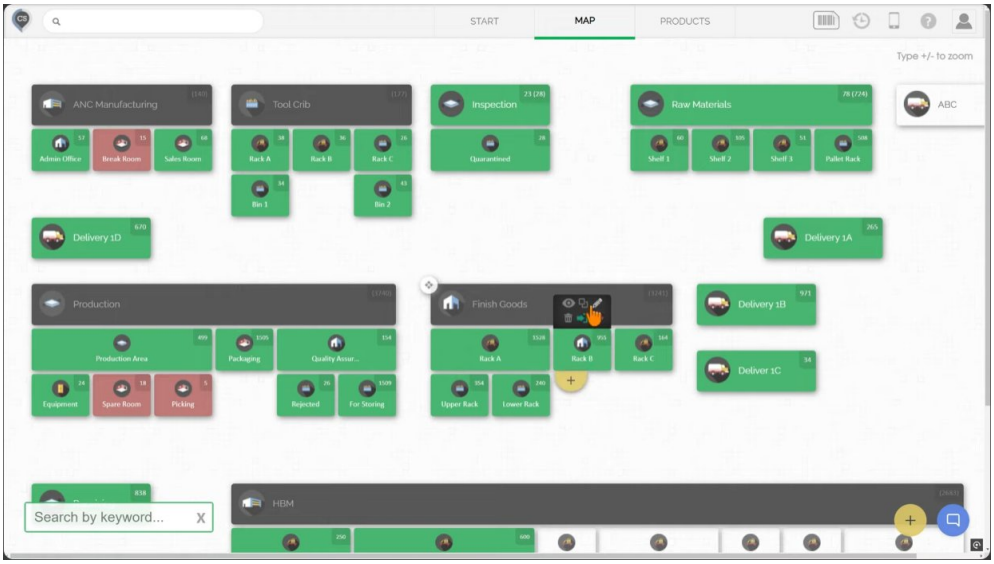


Figure 1.2 CyberStockroom Locations Page

For example, CyberStockroom's own homepage showcases a storage layout with `Closet 1`, `Closet 2`, and `Closet 3`, each containing generic labels such as `Shelf 1` and `Shelf 2`. Although the system provides a visual map, these naming conventions remain unhelpful to anyone unfamiliar with the specific terminology used by the organization. Standing in front of a closet, a user still cannot determine whether `Shelf 1` refers to the top shelf, the bottom shelf, or another shelf altogether without prior explanation.

CyberStockroom's visual approach resembles the hierarchical tree representation that I intend to use. However, CyberStockroom applies this hierarchy to the entire storage environment, whereas my application incorporates accurate physical modeling of floors, furniture, and zones, reserving the tree structure exclusively for items. This distinction makes the system more intuitive and adaptable, enabling its use not only in storage facilities but also in environments such as supermarkets, where it could assist customers in locating items quickly and easily.

1.3 Ikea

Ikea[10] offers a collection of room and furniture planning tools designed primarily for interior visualization and space layout. Although these planners are not intended for inventory management or item tracking, they are relevant to this project because they share similar concepts with the editor components of the proposed application.

However, there are some differences. Ikea's furniture planners rely exclusively on predefined modular components from Ikea's product catalog. In contrast, my application does not have such components. Instead, the user is expected to create any required elements himself using the Shape Editor. This approach offers significantly greater flexibility, allowing users to model elements based on their specific needs rather than being restricted to a fixed set of templates.

[10] https://www.ikea.com/cz/en/planners/

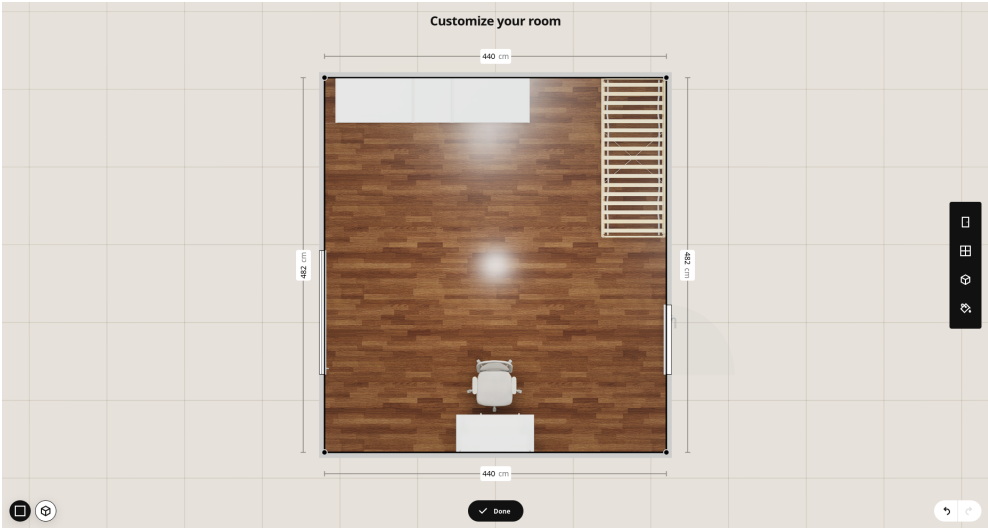


Figure 1.3 Ikea Room Customization Page

Ikea's room planners also differ in their use of both two-dimensional (2D) and 3D to customize floor layouts. Although this provides a richer spatial understanding, it also introduces some usability constraints. Certain interactions are limited to a specific dimension. For instance, furniture placement may be adjusted only in the 3D view, but it is also not possible in the 3D view in the `Customize your room` section of the editor, while wall manipulation is restricted to the 2D view. These mode-dependent constraints can lead to unintuitive behavior, especially when modifying room dimensions after furniture has already been placed. The system attempts to automatically reposition the furniture to maintain a valid placement, but users cannot override the behavior, reducing overall control during editing.

What Ikea's tools illustrate well is the simplicity that I expect my editors to have. The goal of the project is not a full featured CAD system such as Archicad,[11] nor to support very detailed modeling. Instead, the focus is on maintaining a clear and intuitive interface that enables users to configure their floor layouts once and subsequently use the system primarily as an item-tracking tool, similarly to Sortly.

[11] https://www.graphisoft.com/plans-and-products/archicad/

2 Analysis

This chapter provides an analysis for an inventory management system with visual location tracking capabilities. It defines the system's functional and non-functional requirements, specifying the behavior and features the application is expected to support. The chapter also outlines the expected user roles within a production version of the application and presents the essential use cases that illustrate how the system is intended to operate in practice.

The primary purpose of this analysis is to establish a clear and comprehensive understanding of the required system behavior before proceeding to the design phase. By capturing these requirements, the analysis forms the basis for a structured and justifiable system architecture. Furthermore, this chapter establishes the scope of the implemented prototype by identifying which requirements are included and which are deferred, ensuring that the development effort remains focused on the currently most relevant aspects of the system.

2.1 Functional Requirements

This section presents an overview of the functional requirements that define the capabilities the system must provide. It distinguishes the requirements that form the core functionality of the application from those that are deferred for future development, as they are not essential for the basic operation of the prototype. By outlining these requirements, the section clarifies both the expected behavior of the complete system and the specific scope addressed in the implemented prototype.

Requirements marked in red represent features related to multi-user access, role management, or permission handling. Since this project focuses on a single-user environment, these requirements are considered out of scope and will not be implemented.

Requirements marked in teal indicate requirements that are potential future enhancements. They are technically feasible to be reasonably implemented, but will not be included in the current implementation.

FR.1: Users can add, edit, or remove products in inventory.

FR.2: Users can view inventory list.

FR.3: Users can filter and search inventory items.

FR.4: Users can track location of specific items.

FR.5: Users can view detailed information about items.

FR.6: Users can add, edit, or remove shapes.

FR.7: Users can add, edit, or remove furniture.

FR.8: Users can add, edit, or remove floors.

FR.9: Users can view list of shapes and furniture.

FR.10: Users can view list of floors.

FR.11: Users can search and filter shapes and furniture.

FR.12: Users can search and filter floors.

FR.13: Users can move items between different furniture.

FR.14: Users can log in to the system.

FR.15: Users can add, edit, or remove users.

FR.16: Users can create new roles with custom permissions.

FR.17: Users can assign roles to other users.

FR.18: Users can manage permissions for roles.

FR.19: Users can view and manage assigned roles.

FR.20: Users can view list of users.

FR.21: Users can search for specific users and filter them.

FR.22: Users can access a library of shared shapes and furniture.

FR.23: It should be possible to create a first user with all permissions.

FR.24: Users can generate reports on inventory.

FR.25: Users can look through logs of actions taken.

2.2 Non-functional Requirements

This section specifies the non-functional requirements which outline the qualitative properties the system should satisfy. These requirements include properties such as performance, usability, maintainability, and security. The section also includes an explanation of why it may be difficult to fully satisfy all these requirements and why they are mostly expected in a production version of the application.

NR.1: Sensitive information should be encrypted in transit.

NR.2: System should be responsive and fast.

NR.3: Multiple user should be able to use the system simultaneously.

NR.4: The system should be modular, with components logically separated

NR.5: The system should provide a consistent, intuitive, and accessible user interface across all supported devices.

NR.6: The system should be scalable to support increasing amounts of data and users over time.

NR.7: The system should ensure high availability and minimal downtime.

NR.8: All pages should adapt to different screen sizes.

NR.9: The system should be maintainable.

NR.10: The system should be extendable.

NR.11: The system should be resilient against different attacks and errors.

NR.12: The system should be portable.

All the non-functional requirements listed above will be taken into consideration during development. However, as part of this project, we will primarily develop a prototype. Consequently, many of these requirements may not be fully satisfied. Their main purpose is to illustrate how a real production-ready system of this project should behave.

Properly implementing most of these aspects typically requires a significant amount of time and resources. However, they would not bring significant benefits to the scope of this project. Nevertheless, these requirements will guide the development process. For example, they will influence the choice of technologies and the overall system design to align as closely as possible with production standards.

2.3 User Roles

This section identifies user roles relevant to the production version of the application. Each role defines a distinct set of responsibilities and has different permissions within the system. While for implementing the prototype we will focus on a single-user environment, this section helps illustrate how the system could be extended in the future. In addition, this section presents a mapping between the defined functional requirements and the roles that would be authorized to perform them.

Administrator. has unrestricted access to all parts of the application. Their primary responsibility is to keep the system running smoothly. Key responsibilities are the following: Can manage users, inventory, shapes, furniture, floors, and permissions. Can add/edit/remove any user and assign roles. Can modify inventory and perform all operations.

Warehouse Worker. In reality, it can be an inventory clerk, forklift operator, or another person who manages the inventory. His responsibility in the system is to add items to the zones for the floor he works at, update locations, etc. Key responsibilities are the following: Update stock, view inventory, track locations. Cannot modify shapes, furniture, floors, and users.

Observer. Can be a customer, an executive, or even a stakeholder. They have read-only access to the inventory, so they can keep track of the inventory. The customer in a supermarket can use the system to find a specific location of an item he is looking for, etc. Key responsibilities are the following: Browsing products without being allowed to make any modifications.

Warehouse Manager. It is a person who manages warehouse workers and makes sure that inventory is properly managed. In case of inventory mismatch, he can also make adjustments in the inventory himself. Key responsibilities are the following: Manages Warehouse Workers assigned to them. Can perform inventory operations. Cannot modify users not assigned. Cannot modify shapes, furniture, and floors.

Furniture Editor. It can be, for example, an interior architect. His main job is to prepare the furniture. That includes creating the main frame of the furniture, but most importantly, he should choose specific zones within the furniture into which it should be possible to place items, for example, a drawer of a cupboard or a shelf. His job is also to create the shapes that he wants to use to create the furniture. Key responsibilities are the following: Can add/edit/remove furniture and shapes. Cannot modify users, inventory, and floors.

Floor Editor. Can be, for example, a facilities planner. His main job is to create the layout of the floor or room. That means he should create the corners and the walls representing the real life counter part and then he should place furniture according to the real life placement. Key responsibilities are the following: Can add/edit/remove floors. Cannot modify users, inventory, shapes, and furniture.

Editor Manager. Can be some sort of design supervisor. His responsibility is to supervise all the editors. He can make sure that the furniture and floor layouts are consistent with the real world and if he needs he has a possibility to make adjustments. Key responsibilities are the following: Can add/edit/remove shapes, furniture, and floors. Can manage users assigned to him. Cannot modify users not assigned to him. Cannot modify inventory. Can assign the editor roles.

All the above listed roles represent the set of roles that would be expected in the fully featured production version of the application. However, for the purpose of this project, the implemented system will work with a single user role that has unrestricted access to all functionality. Although this role effectively represents the administrator, it will not be explicitly modeled as such.

The decision to implement only one role is deliberate as it allows me to focus on the core functionality and the more relevant aspects of the system, rather than implementing logic that is really common within most modern applications. By not implementing the multi-role permission structure, the project can concentrate on essential features without being burdened with additional complexity.

Functional Requirement	Admin	Warehouse Worker	Observer	Warehouse Manager	Furniture Editor	Floor Editor	Editor Manager
FR.1	✓	✓		✓			
FR.2	✓	✓	✓	✓			
FR.3	✓	✓	✓	✓			
FR.4	✓	✓	✓	✓			
FR.5	✓	✓	✓	✓			
FR.6	✓				✓		✓
FR.7	✓				✓		✓
FR.8	✓					✓	✓
FR.9	✓				✓		✓
FR.10	✓					✓	✓
FR.11	✓				✓		✓
FR.12	✓					✓	✓
FR.13	✓	✓		✓			
FR.14	✓	✓	✓	✓	✓	✓	✓
FR.15	✓			✓			✓
FR.16	✓						
FR.17	✓						✓
FR.18	✓						
FR.19	✓			✓			✓
FR.20	✓			✓			✓
FR.21	✓			✓			✓
FR.22	✓				✓	✓	✓
FR.23							
FR.24	✓			✓			
FR.25	✓			✓			✓

Table 2.1 Mapping of functional requirements to relevant user roles

2.4 Use Cases

This section presents the primary use cases that illustrate how users are expected to interact with the system. The use cases provide a complementary perspective to the functional requirements. While the requirements specify what the system must offer, the use cases describe these functionalities from the user's

point of view. Each use case outlines a sequence of user actions along with the corresponding system responses, which offers us a practical view of the intended behavior. In addition, a flow chart is included with every use case that visually represents the sequence of actions, modeled using the unified modeling language (UML) [4].

2.4.1 View the inventory list

Accessing the inventory list is one of the key interactions of the system and corresponds to requirement FR.2. The goal of this use case is to allow the user to view all available inventory items. Diagram 2.1 illustrates the use case from the perspective of Warehouse Worker, although the interaction is identical for all other relevant user roles.

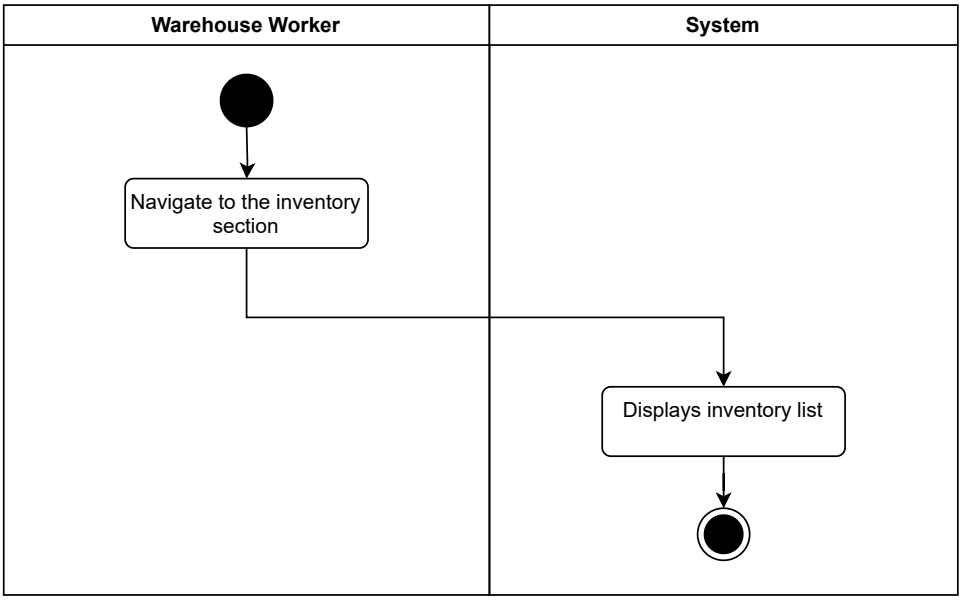


Figure 2.1 Flowchart for viewing the inventory list

Initial assumption:

- Logged in user is one of the relevant user roles.

Normal flow:

1. The user navigates to the inventory section by clicking the appropriate menu item, button, or entering the corresponding uniform resource locator (URL).
2. The system displays the complete list of inventory items in a default order (e.g. alphabetical or by category).

What can go wrong:

- Inventory fails to load due to a network or server error. In this case, the system displays an appropriate error message informing the user of the issue.

System state on completion:

- The user is presented with the current inventory list. If loading fails, the user is shown an appropriate error message.

2.4.2 Track the location of a specific item

Tracking the location of an item allows users to quickly determine where a particular item is located. This interaction corresponds to the functional requirement FR.4. Diagram 2.2 illustrates the use case from the perspective of Warehouse Worker, although the interaction is identical for all other relevant user roles.

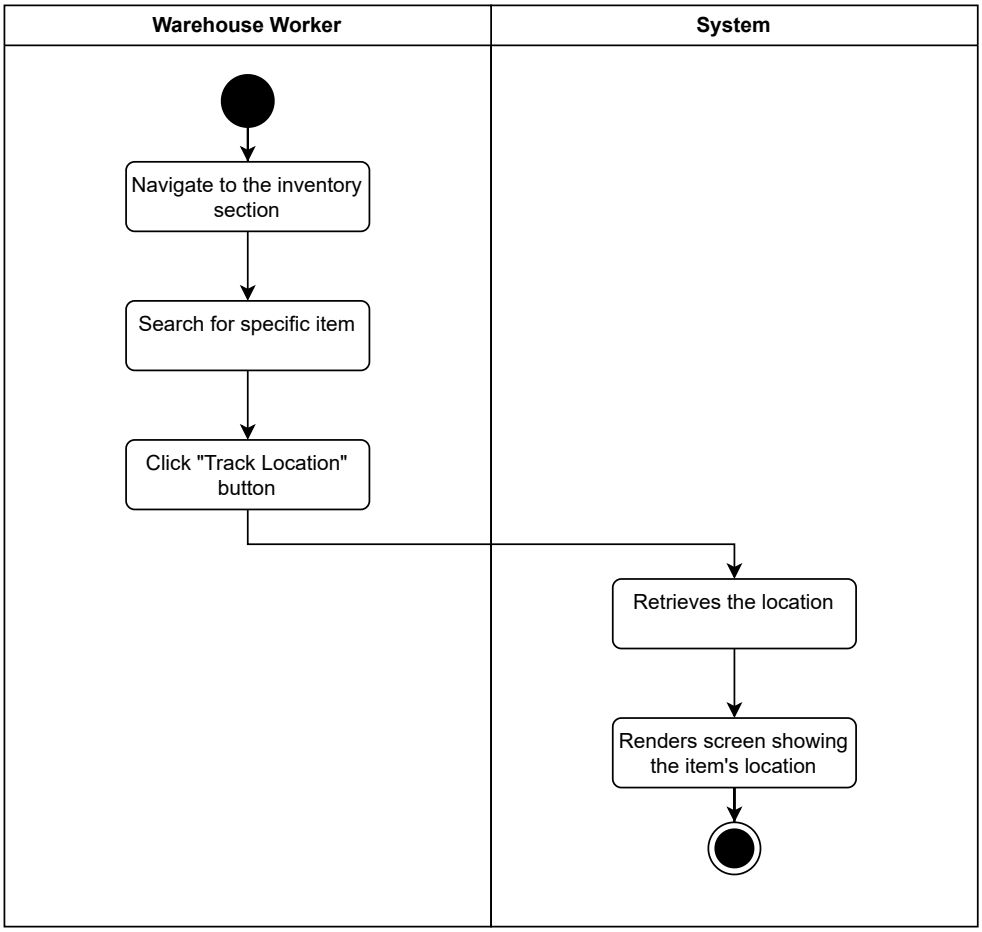


Figure 2.2 Flowchart for tracking the location of a specific item

Initial assumption:

- Logged in user is one of the relevant user roles.

Preconditions:

- The item being searched for exists in the inventory database.

Normal flow:

1. The user navigates to the inventory section.
2. The user locates the desired item item using the search bar, applying filters, or scrolling through the inventory list.
3. The user presses the button to track the location of the item.
4. The system retrieves information about where the item is stored.
5. The system displays an interactive screen showing the item's location on the corresponding floor, highlighting the specific furniture in which the item is placed.

What can go wrong:

- The system cannot retrieve the location of the item due to a network or server error. In such a case, an error message is displayed to the user.

System state on completion:

- The user sees an interactive representation of the item's location, or an appropriate message is displayed if the location could not be retrieved.

2.4.3 Add, edit, or remove a shape

Managing shapes is required to allow relevant user roles to maintain geometric templates used for furniture definitions within the application. This functionality corresponds to requirement FR.6. Diagrams 2.3 2.4 2.5 illustrate the use case from the perspective of Furniture Editor, although the interaction is identical for all other relevant user roles.

Initial assumption:

- Logged in user is one of the relevant user roles.

Preconditions:

- The user has navigated to the shape management section.

Add flow:

1. The user clicks the button to add a new shape.
2. The system opens the shape editor and displays a list of existing shapes that can be used as a base.
3. The user constructs a new shape by combining or modifying existing shapes.
4. The user assigns a name to the new shape.
5. The user clicks the save button.
6. The system validates the input data.
7. The system saves the new shape.

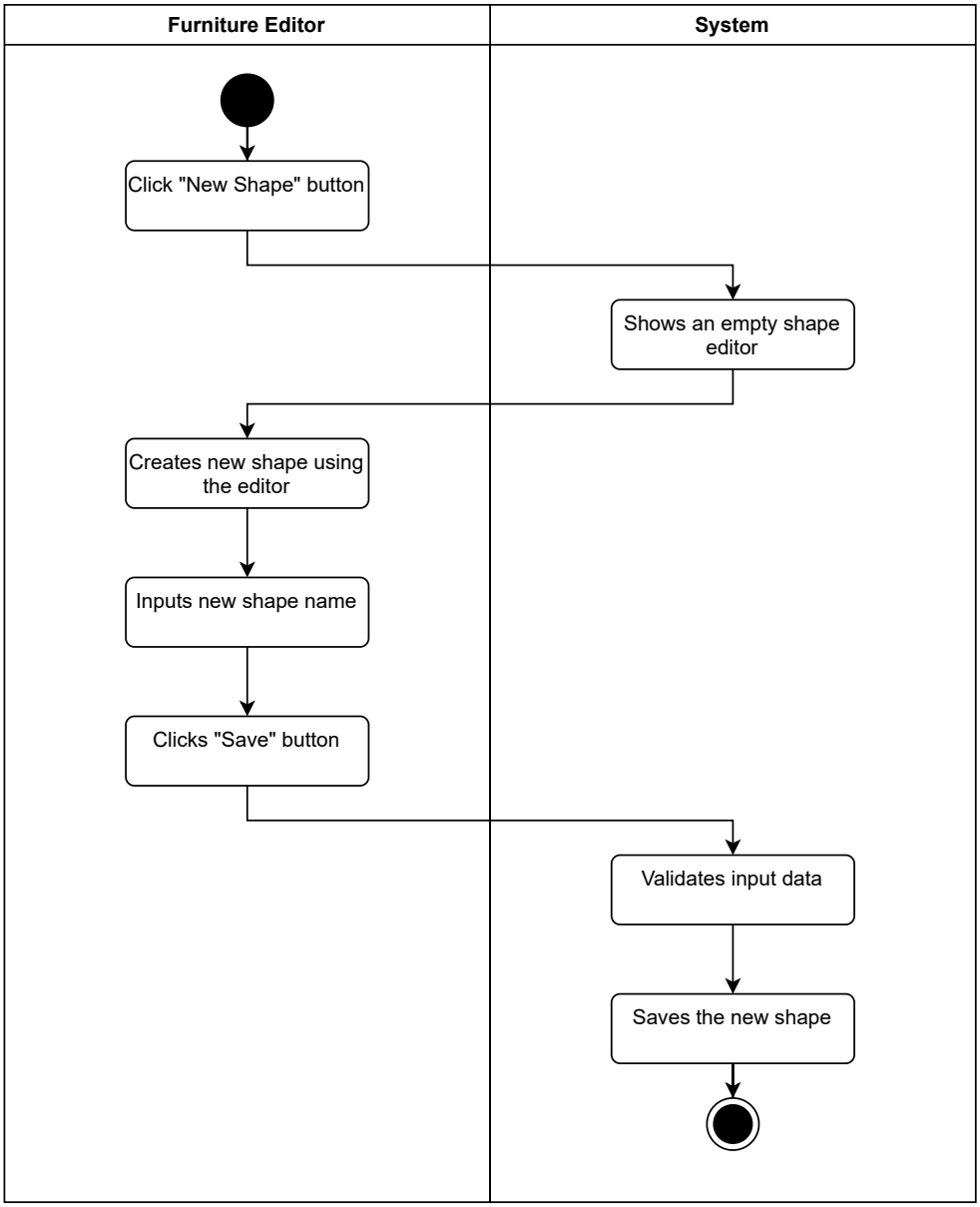


Figure 2.3 Flowchart for adding a new shape

Edit flow:

1. The user finds the existing shape from the list that he wants to edit.
2. The user clicks the Edit button.
3. The system shows the editor with the selected shape preloaded.
4. The user modifies the shape as needed.
5. The user may also change the name of the shape.
6. The user clicks the save button.

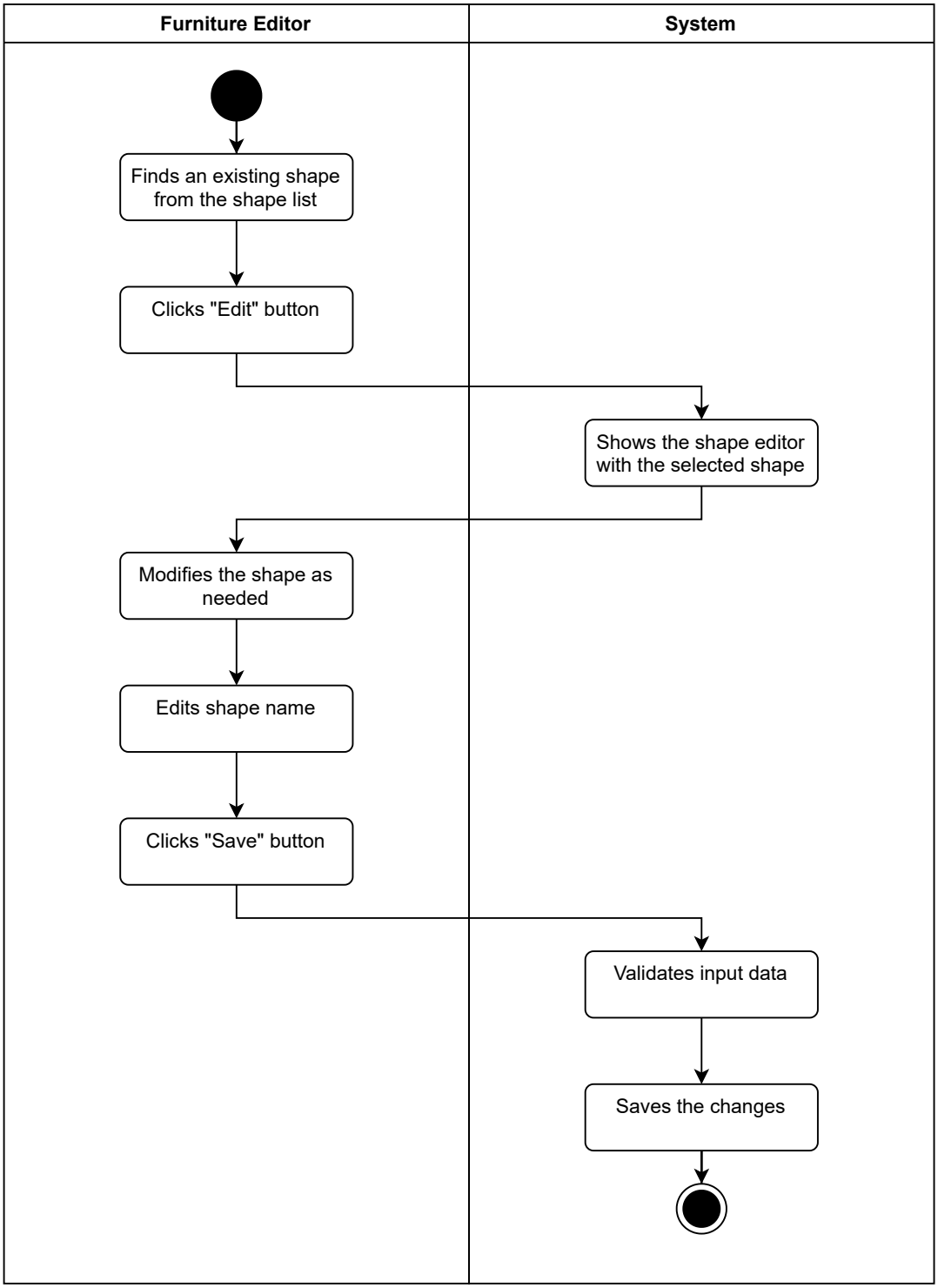


Figure 2.4 Flowchart for editing an existing shape

7. The system validates the input data.
8. The system saves the changes to the shape.

Remove flow:

1. The user finds the existing shape from the list that he wants to remove.

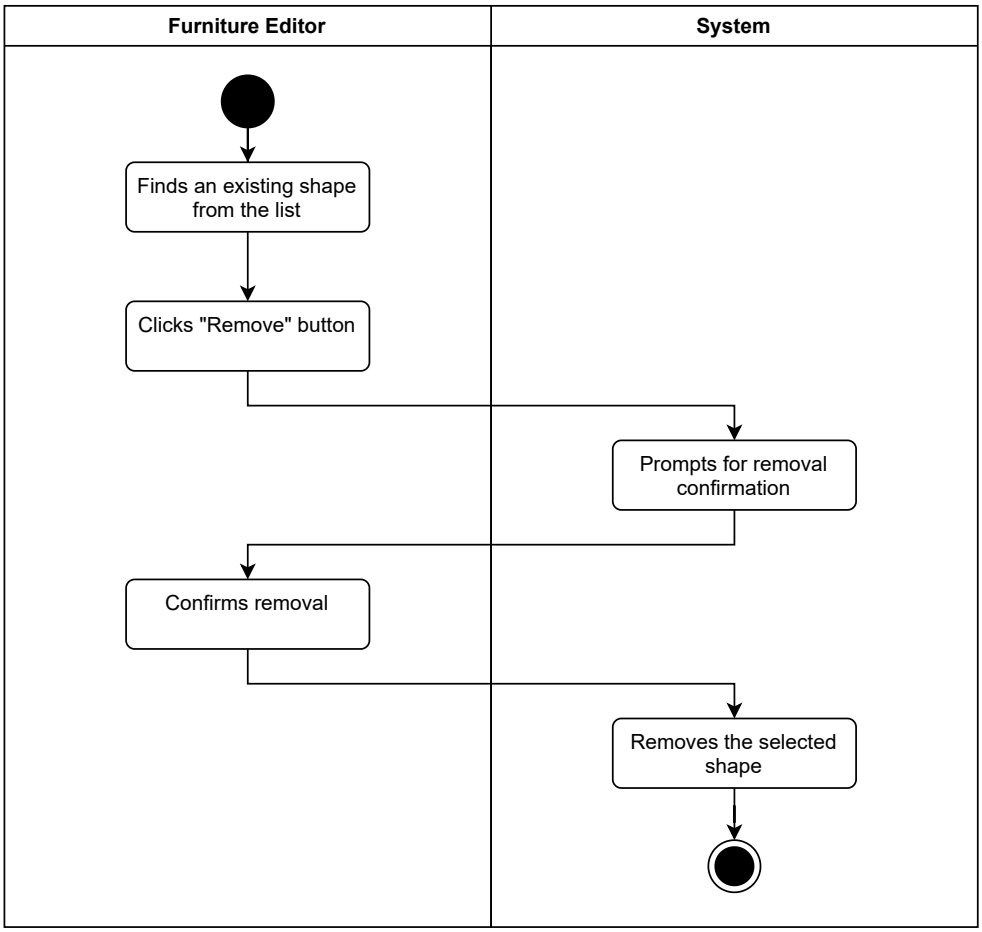


Figure 2.5 Flowchart for removing an existing shape

2. The user clicks the remove button.
3. The system asks the user to confirm the removal.
4. The user confirms the removal.
5. The system removes the shape.

What can go wrong:

- The user enters invalid data (e.g. missing required fields, incorrect format). An error message is displayed.
- The system cannot save or remove a shape due to a server or network error. The user is informed via an error message.

System state on completion:

- A shape is successfully added, modified, or removed, or the user is informed of an error using an appropriate message.

2.4.4 Add a furniture

Creating new furniture is necessary for defining objects that can be positioned on floors and used as containers for inventory items. This enables users to place items within specific furniture units, supporting more precise and structured location tracking. This functionality corresponds to requirement FR.7. Diagram 2.6 illustrates the use case from the perspective of Furniture Editor, although the interaction is identical for all other relevant user roles.

Initial assumption:

- Logged in user is one of the relevant user roles.

Preconditions:

- The user has navigated to the furniture management section.

Normal flow:

1. The user clicks the button to add new furniture.
2. The system opens the furniture editor and displays the available shapes.
3. The user uses the editor to create the structure of the furniture using the provided shapes.
4. The user uses the editor to create `placeable` areas for the furniture using the provided shapes.
5. The user chooses a furniture name using the name input
6. The user clicks the save button.
7. The system asks the user to choose a top-down view of the furniture.
8. The user selects the top-down view of the furniture.
9. The user clicks the confirmation button.
10. The system validates the input data.
11. The system saves the new furniture.

What can go wrong:

- The user enters invalid data (e.g. missing required fields, incorrect format). An error message is displayed.
- The system cannot save the furniture due to a server or network error. The user is informed via an error message.

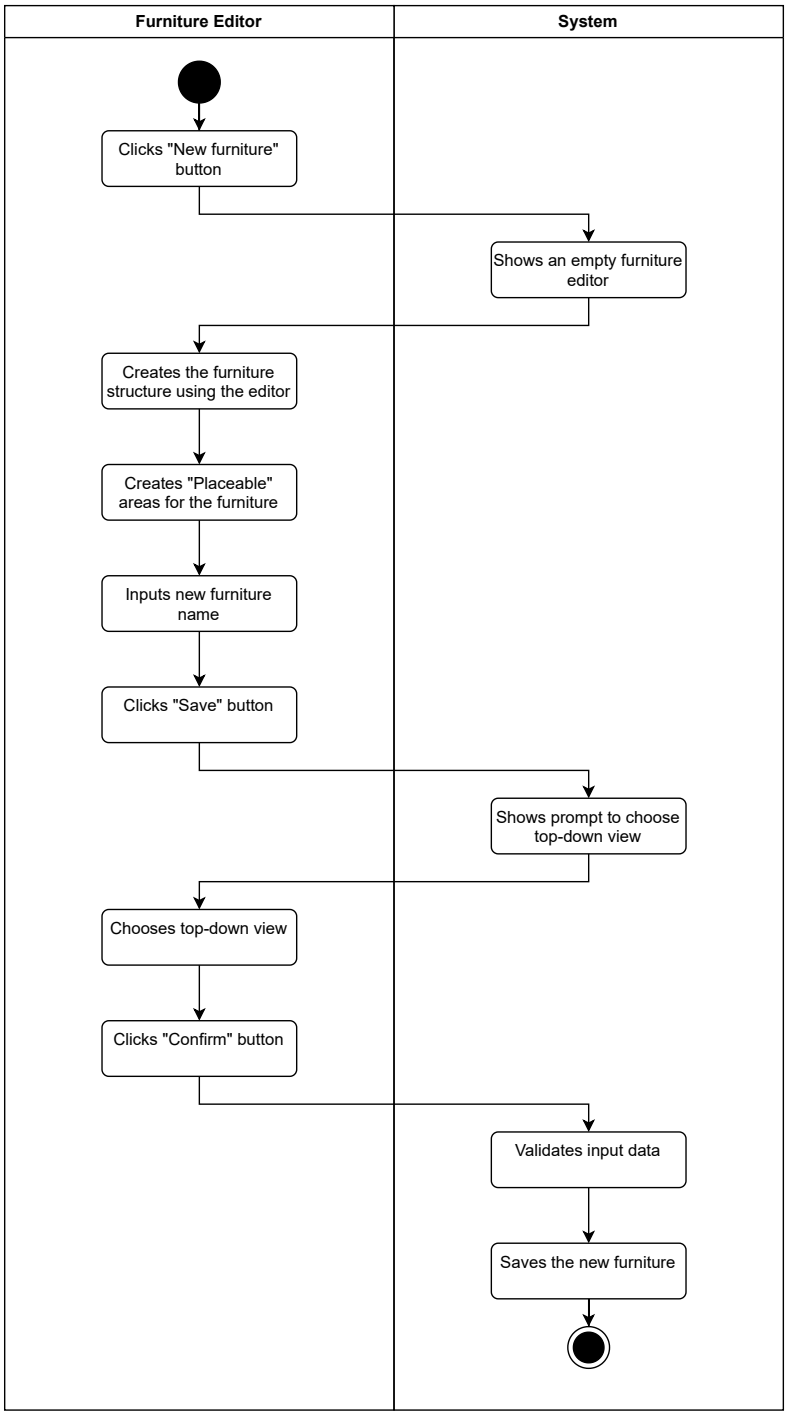


Figure 2.6 Flowchart for adding new furniture

System state on completion:

- The new furniture is saved and displayed in the furniture list, or an appropriate error message is shown if the action fails.
- The furniture is successfully added and appears in the furniture list, or the user is informed of an error using an appropriate message.

2.4.5 Add a floor

Adding a new floor allows the system to model a physical space where furniture can be placed. By defining floors, users can then position furniture accurately within it, which in turn enables precise tracking of inventory items by associating them with the placed furniture. This functionality corresponds to requirement FR.8. Diagram 2.7 illustrates the use case from the perspective of Floor Editor, although the interaction is identical for all other relevant user roles.

Initial assumption:

- Logged in user is one of the relevant user roles.

Preconditions:

- The user has navigated to the floor management section.

Normal:

1. The user clicks the button to add a new floor.
2. The system opens the floor editor and displays the available furniture.
3. The user toggles on the corner placement by pressing the corner button
4. The user clicks within the editor to define the corners of the floor.
5. The user toggles off the corner placement by pressing the corner button
6. The user clicks two corners to create a wall of the floor.
7. The user adds furniture to the floor using the editor.
8. The user chooses a floor name using the name input
9. The user clicks the save button.
10. The system validates the input data.
11. The system saves the new floor.

What can go wrong:

- The user enters invalid data (e.g. missing required fields, incorrect format). An error message is displayed.
- The system cannot save the floor due to a server or network error. The user is informed via an error message.

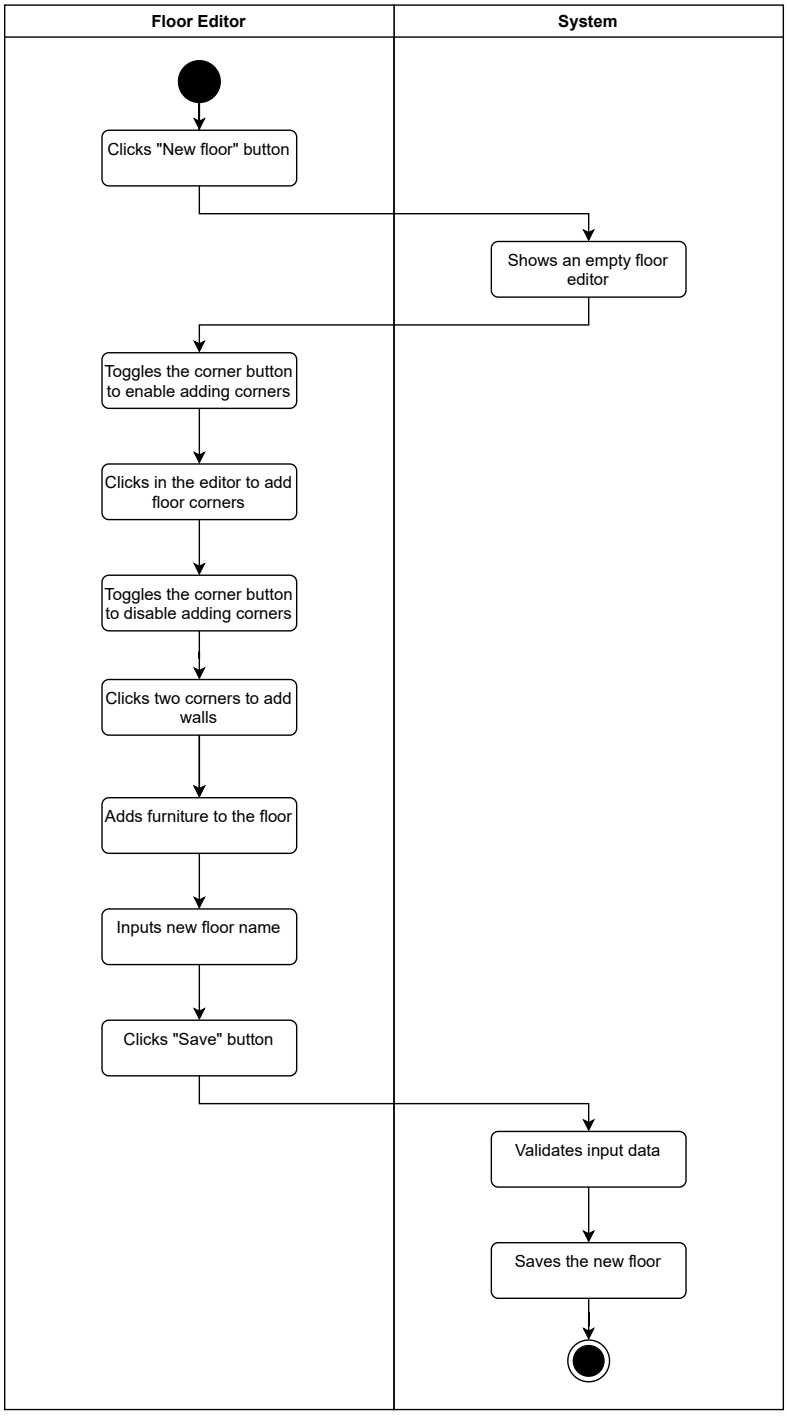


Figure 2.7 Flowchart for adding a new floor

System state on completion:

- The floor is successfully added and appears in the floor list, or the user is informed of an error using an appropriate message.

3 Design Documentation

This chapter outlines the overall design of the system. It explains the system architecture using a C4 model. It also presents the expected data structures, talks about the chosen technologies, and includes mockups that show a visual representation of several application pages.

The chapter is divided into several sections. It begins with the C4 model and moves through its layers. It introduces the system context diagram, the container diagram, and the component diagram. Each diagram shows how the system is decomposed into separate elements and how those elements relate to each other. Following this, the domain model is presented, detailing the data entities of the system, their attributes, and the relationships between them. The next section discusses the technologies chosen for development, provides a justification for each selection, and explains why some alternatives were not chosen. Finally, the chapter concludes with visual mockups, offering an initial impression of the intended user interface and demonstrating how key parts of the application may appear in the final system.

3.1 C4 Model

This section presents the C4 model used to describe the architecture of the system, illustrating its decomposition at multiple levels. It starts with a visual of the system's high-level context. Then, it displays its internal containers, followed by showing the components within those containers.

The purpose of this section is to clearly explain how the system is organized, how its parts interact, and the roles of the different components.

3.1.1 System Context Diagram

The diagram 3.1 shows the system context of the application. As we do not use any external services, our system consists of a single system, and this diagram mainly shows all the actors acting upon the system. Although, we show all the actors here, it is mainly displayed to show what actors would be expected in some final production version of the application. Within the scope of this project we will assume only a single actor which within this diagram would be represented as the administrator. However, it will not be specifically modeled as such within the application.

We can see that the primary system is used to control the entire application. It is used to manage inventory, floors, furniture, shapes, users, etc. It also provides the user interface, handles security, among other things.

Our system consists of seven actors:

- Administrator
- Warehouse Worker
- Warehouse Manager
- Furniture Editor

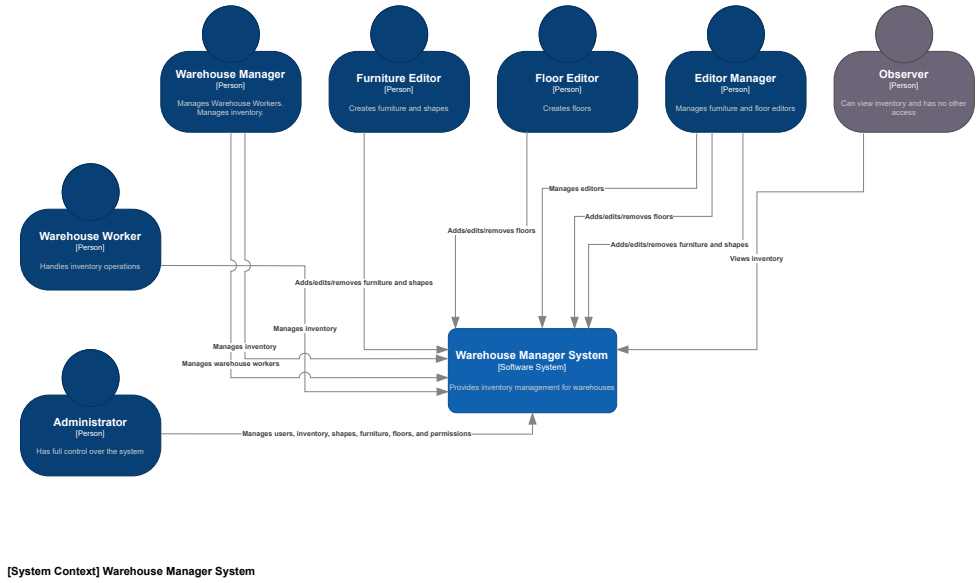


Figure 3.1 System Context Diagram

- Floor Editor
- Editor Manager
- Observer

3.1.2 Container Diagram

The diagram 3.2 illustrates that the system is a microservices-based architecture since it is separated into multiple services. The services are then separated from the application running in the browser using an application programming interface API gateway, which mediates communication between the services. Every service is responsible for managing some aspect of the application, such as generating the user interface, managing floors, etc.

However, selecting a microservice architecture in this context may not be the most optimal choice. The primary motivation for using it was to gain hands-on experience with the technologies and concepts associated with microservices, rather than relying on a more conventional three-tier architecture.

In retrospect, the microservices approach may not have been the best fit because several of the defined services behave more like components of a monolithic system. In a real-world scenario, their dependency would likely require them to be developed and maintained by a single team. Furthermore, this architecture can introduce unnecessary operational overhead for deployment in a home or small-business environment, where applications are typically expected to run on a single machine. The need to operate multiple services increases resource consumption and may require more capable hardware, which can be impractical in such settings.

Despite that, it still makes the system more scalable and can be easier to extend with new functionality.

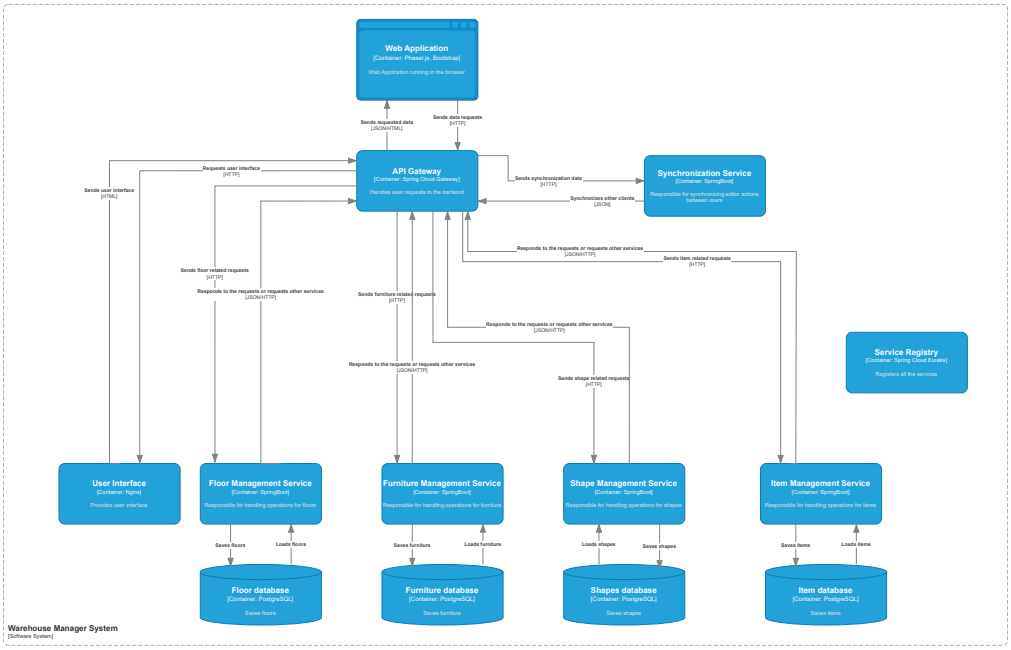


Figure 3.2 Container Diagram

Note that even though some of the services in the diagram communicate directly with each other, for simplicity, it is shown that they communicate through the API Gateway.

The following section describes the system's components, their respective responsibilities, and how the services communicate.

Web Application (Frontend) Is a multi-page application running in the user's browser. All of the frontend's files are served through the API Gateway, which forwards the request to the server hosting the files. The frontend communicates with backend services exclusively by sending hypertext transfer protocol (HTTP) requests to the API Gateway, which returns JavaScript object notation (JSON) responses.

API Gateway Component responsible for forwarding all calls to the appropriate backend microservices. It acts as a single public entry point into the system. Helps hide internal service structure from the clients. Will be implemented using Spring Cloud Gateway.

User Interface Server Component that hosts all the frontend assets. It has no real application logic and is only responsible for serving files. It is accessed through API Gateway, which is responsible for redirecting all browser redirects to this component.

Service Registry All backend microservices register themselves with the registry, allowing other services to dynamically discover them. This eliminates the need for hard-coded service addresses allowing us to reference a service based on a key, additionally simplifying inter-service communication. Because services

discover each other at runtime, instances can be added, removed, or relocated without affecting the rest of the system. This enables easy scaling and fault tolerance, as we can deploy multiple replicas of the same service. Will be implemented using Spring Cloud Eureka

Management Components All of these services are responsible for handling operations in their associated domain. They are responsible for saving and loading data into their respective database.

Within our domain, there are dependencies between these services. The communication between these services is made direct, to not over-crowd the API Gateway, despite it requiring to have the services less isolated on a network. This differs from the diagram above where for simplicity it is shown that all the services communicate through the API Gateway.

List of the management components:

- **Floor Management Service**
- **Furniture Management Service**
- **Shape Management Service**
- **Item Management Service**

Synchronization Service This component will not be implemented within this project but is here to show some additional services that can be future enhancements to the application. The component is responsible for handling the synchronization of multiple user's editors in case they are working on the same shape, furniture, or floor. The service should guarantee that the synchronization is almost instantaneous to make it comfortable for the editors to use, and additionally, it should prevent having inconsistencies in data if multiple users work on the same element.

Databases They are used to save all the data from their associated service.

List of databases:

- **Floor Database**
- **Furniture Database**
- **Shape Database**
- **Item Database**

3.1.3 Component Diagrams

This section shows the component diagrams of the C4 model[3]. It provides a more detailed view of the internal structure of the system's containers. Its purpose is to show how the components are organized and how they should be implemented, as these components are typically reflected in the system's code.

Web Application(Frontend)

The component illustrated in Figure 3.3 represents the complete frontend of the application running in the user's browser. All communication with the backend is required to go through the API Gateway, making this component as isolated as possible.

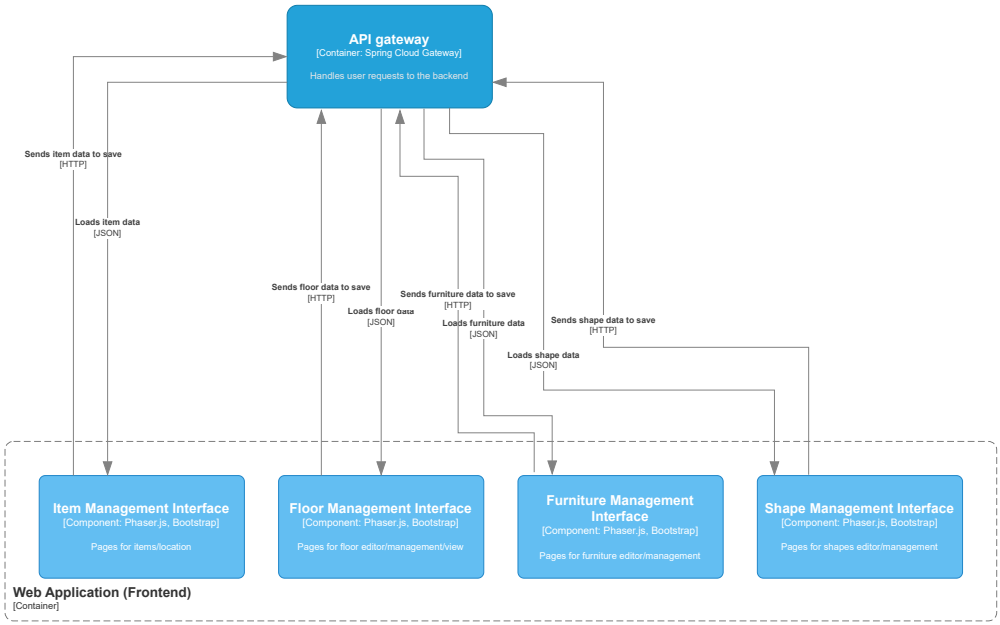


Figure 3.3 Component diagram for the web application(frontend)

All other components correspond to the different functional sections of the application:

Item Management Interface Is responsible for providing pages for listing the items and tracking the item location. It should also allow creating/updating/deleting items

Floor, Furniture, and Shape Interfaces Each domain should contain: a list page where the user can see a list of all elements of the domain and additionally delete them, and an editor for creating and modifying the associated elements. In addition, there should be a viewer where users can view their created floor and inspect inside the furniture instances and subsequently also move existing items into the furniture instances. This will be made possible by showing a list of all existing items that the user can drag into a specific zone within the furniture instance.

The editors should have some basic features that are normally expected within an editor, such as resizing, moving, etc.

Management Services

All of these services work in a similar manner. Figure 3.4 shows the component diagram for Shape Management Service, but the diagram is the same for the other services. The figure illustrates that all the services receive a request from the API

Gateway. The handler is then responsible for sending the data to the validator to check if the data is in the correct format. The data is then sent to the query interface that is responsible for saving/loading data from the database. Finally, the response is sent from the handler back to the API Gateway.

In the context of Spring, which will be used for these services. The Request Handler is represented by a Rest Controller, which also immediately sends the incoming Entity to a validator. The Database Query Interface in Spring is represented by a Repository.

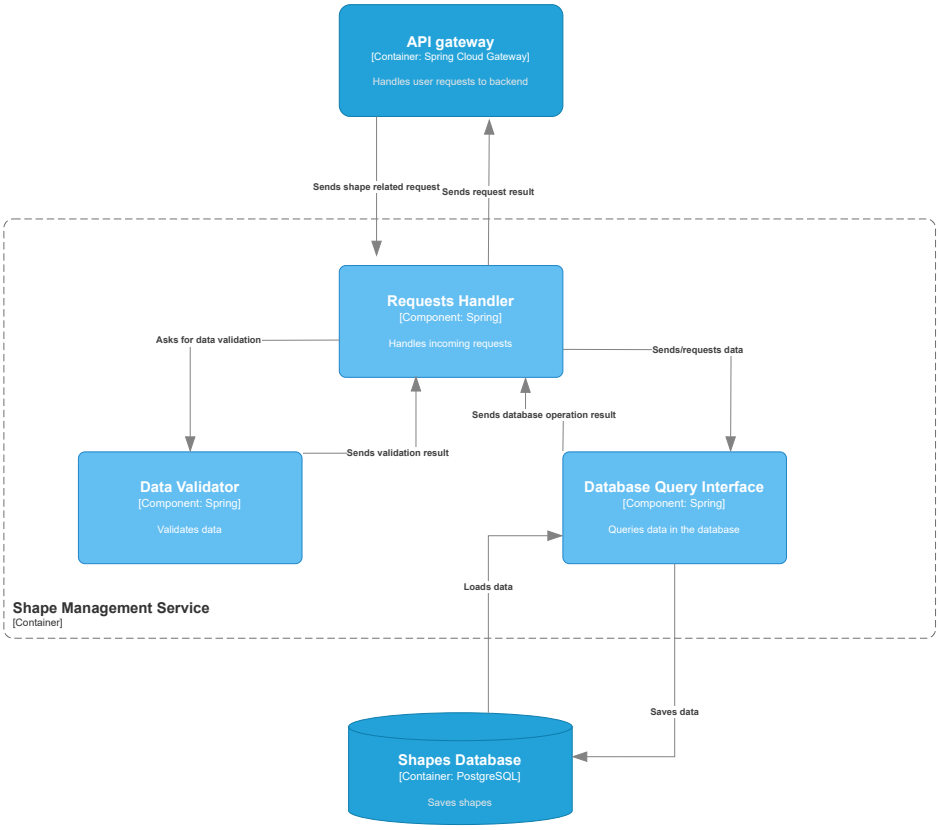


Figure 3.4 Component diagram for the shape management service

Synchronization Service

The figure 3.5 illustrates that because the service requires real-time communication between clients, the service uses a WebSocket which enables the service to create a bidirectional tunnel through which the client and the service can communicate almost in real time. In addition, the service has a request handler that is responsible for mediating the connection. Additionally, there is a component to save the editor contexts and other required data in memory. It enables faster access than using a database which this service does not need as there is no requirement to persist data, as the connections to this service are not permanent and all data should get deleted after the connection is terminated.

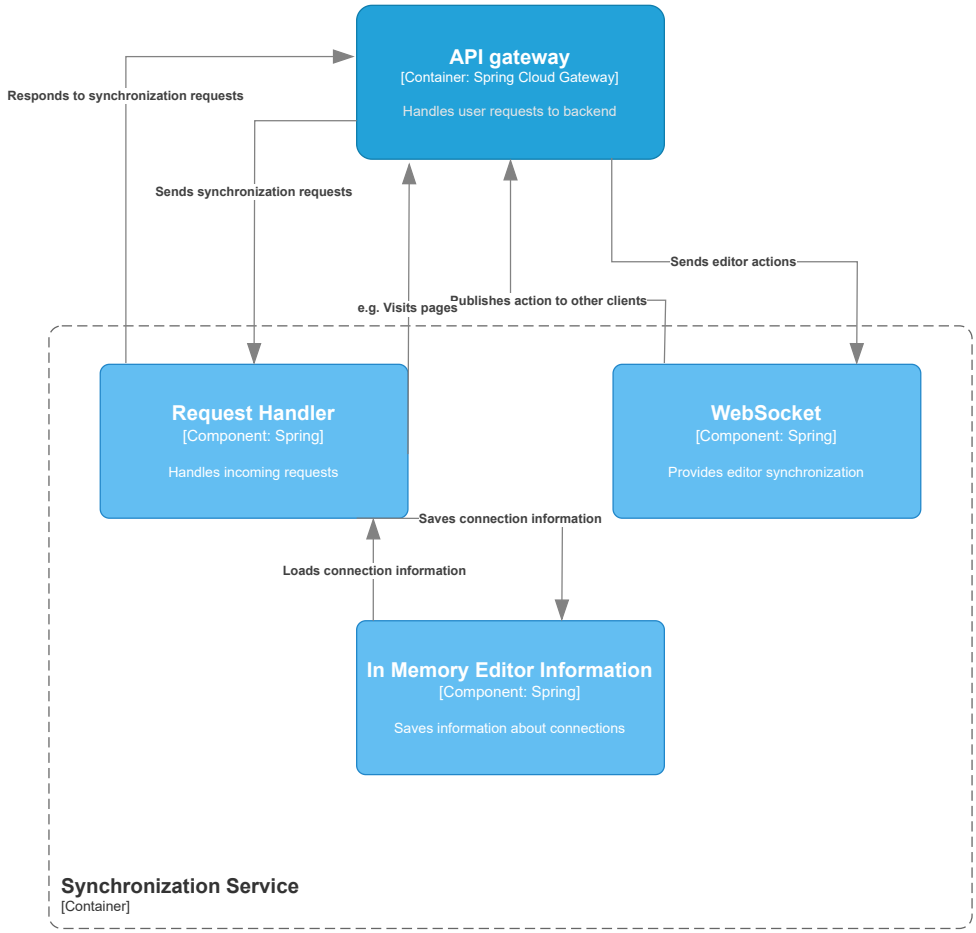


Figure 3.5 Component diagram for the synchronization service

3.2 Domain Model

The diagram 3.6 shows the data that will be used in the application and the expected relationships between the specific classes. This diagram can be divided into four separate sections: shape, furniture, floor, and item. As we will not be implementing the multi-user environment, the user section is not accounted for in the domain model. The model is modeled using UML[4].

To support traceability and historical consistency of domain entities, the domain model incorporates explicit versioning for selected classes. Versioning is required to preserve earlier states of entities while allowing for controlled evolution of their definitions over time. This is particularly important for reusable components, such as shapes and furniture, where changes should not invalidate existing references or previously created structures.

The versioning strategy used in this project follows the Type 2 Slowly Changing Dimension approach as described by Kimball and Ross in *The Data Warehouse Toolkit* [5]. In this approach, modifications to an entity result in the creation of a new version rather than overwriting existing data. Each version is retained as a separate record, allowing full historical tracking while clearly identifying the currently active version.

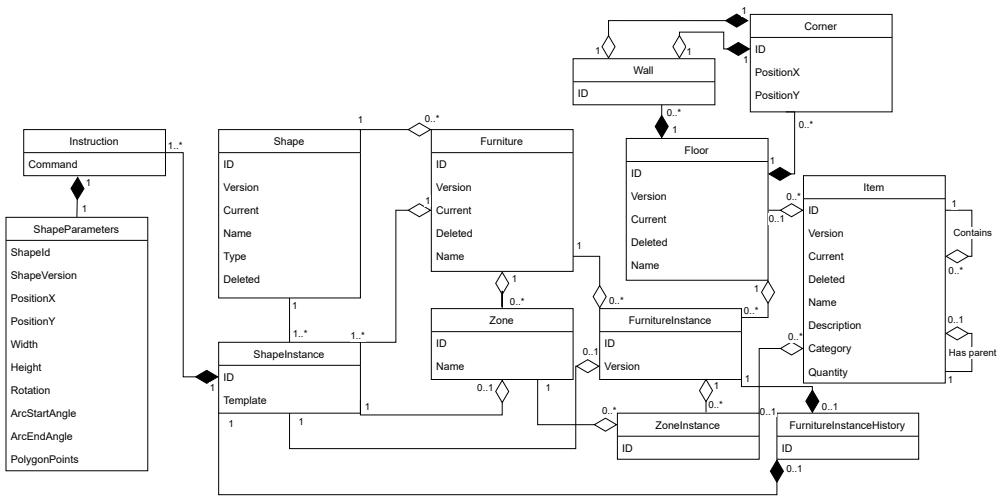


Figure 3.6 Domain model diagram

Shapes

The following data entities are entities that should be part of the Shape Management Service. All of these entities are related to shapes.

Shape Represents the base definition of a shape within the warehouse management system.

Shape includes the following attributes:

- **ID** Unique identifier of the shape shared across all versions
- **Version** Version timestamp, modifications provide new versions
- **Current** Indicates if this is the latest version of the shape
- **Name** Name of the shape
- **Type** The specific type of the shape (e.g. Container, Rectangle, etc.)
- **Deleted** Indicates that the shape is no longer allowed to be used to create new instances and could be used to trigger some fallback rendering behavior.

Shape is involved in the following relationships:

- A defined shape is referenced by one or more ShapeInstances.
- Referenced by 0 or more Furniture definitions

ShapeInstance Represents a concrete instance of a shape within the warehouse, bound to a specific version

ShapeInstance includes the following attributes:

- **ID** Unique identifier of the shape instance

- **Template** Indicates if this instance works as a reusable template

ShapeInstance is involved in the following relationships:

- ShapeInstance references exactly a single Shape
- Is top-down view instance of 0 or 1 FurnitureInstance
- Composes 0 or 1 Furniture
- Contains 1 or more Instruction describing how to construct the instance
- Referenced by 0 or 1 FurnitureInstanceHistory
- Shape of 0 or 1 Zone

Instruction Represents a single command used to construct a ShapeInstance

Instruction includes the following attributes:

- **Command** Type of command (e.g., `CREATE_RECTANGLE`)

Instruction is involved in the following relationships:

- Each instruction belongs to a single ShapeInstance
- Contains single ShapeParameters

ShapeParameters Encapsulates all parameters required by an Instruction

ShapeParameters include the following attributes

- **ShapeId** ID of the shape the command associated with these parameters represents
- **ShapeVersion** Version of the referenced shape
- **PositionX**, **PositionY** Coordinates of the shape
- **Width**, **Height** Size of the shape
- **Rotation** Rotation angle
- **ArcStartAngle**, ArcEndAngle Start and end angle of arc shape
- **PolygonPoints** Coordinates for polygon vertices

ShapeParameters are involved in the following relationships:

- Referenced by exactly one Instruction

Furniture

The following data entities are entities that should be part of the Furniture Management Service. All of these entities are related to furniture.

Furniture Represents a definition of a piece of furniture

Furniture includes the following attributes:

- **ID** Unique identifier of the furniture
- **Version** Version timestamp, modifications provide new versions
- **Current** Indicates whether this version is the latest
- **Deleted** Indicates that it is not possible to create new instances of this furniture
- **Name** Name of the furniture

Furniture is involved in the following relationships:

- References a single Shape as its top-down view
- References 1 or more ShapeInstance that compose the furniture structure
- References 0 or more Zones where objects can be placed
- Referenced by 0 or more FurnitureInstances

Zone Represents a named region within a Furniture definition where items may be placed

Zone includes the following attributes:

- **ID** Unique identifier of the zone
- **Name** Name of the zone

Zone is involved in the following relationships:

- References a single ShapeInstance
- Belongs to a single Furniture definition
- Referenced by zero or more ZoneInstances

FurnitureInstance Represents a concrete instance of a Furniture placed in the warehouse

FurnitureInstance includes the following attributes:

- **ID** Unique identifier of the FurnitureInstance
- **Version** Version timestamp, modifications create new versions

FurnitureInstance is involved in the following relationships:

- References a single specific ShapeInstance representing the placement
- Contains zero or more ZoneInstances
- Placed within a single Floor
- References a single Furniture
- Referenced by 0 or 1 FurnitureInstanceHistory

FurnitureInstanceHistory Captures the evolution of FurnitureInstances

FurnitureInstanceHistory includes the following attributes:

- **ID** Unique identifier of the History

FurnitureInstanceHistory is involved in the following relationships:

- References a single specific ShapeInstance representing the placement
- History of a specific FurnitureInstance of a given version

ZoneInstance Represents an instance of a Zone within a FurnitureInstance

ZoneInstance includes the following attributes:

- **ID** Unique identifier of the ZoneInstance

ZoneInstance is involved in the following relationships:

- Contains zero or more Items
- Each ZoneInstance references a single Zone
- ZoneInstance belongs to a single FurnitureInstance

Floor

The following data entities are entities that should be part of the Floor Management Service. All of these entities are related to floors.

Floor Represents a level within a building or a specific room

Floor includes the following attributes:

- **ID** Unique identifier of the floor
- **Version** Version timestamp, modifications provide new versions
- **Current** Indicates whether this version is the latest version.
- **Deleted** Indicates that it is no longer possible to use this floor
- **Name** Name of the floor

Floor is involved in the following relationships:

- Contains any number of FurnitureInstance
- Contains any number of Wall
- Contains any number of Corners

Wall Represents a wall between two Corners within a Floor

Wall includes the following attributes:

- **ID** Unique identifier of the Wall

Wall is involved in the following relationships:

- Consist of Corner as start and the end of the wall
- Is part of a single Floor

Corner Represents a corner within a floor

Corner includes the following attributes:

- **ID** Unique identifier of the corner
- **PositionX** X-coordinate of the corner
- **PositionY** Y-coordinate of the corner

Corner is involved in the following relationships:

- Serves as an ending for Walls
- Is part of a single Floor

Item

The following data entity should be part of the Item Management Service. It represents the items.

Item Represents an object user wants to keep track of, such as supplies or tools

Item includes the following attributes:

- **ID** Unique identifier of the item
- **Version** Version timestamp, modifications provide new versions
- **Current** Indicates whether this version is the latest version.
- **Deleted** Indicates that it is no longer possible to use this item
- **Name** Name of the item
- **Description** Long description of the item
- **Category** Category into which user wants to group similar objects together
- **Quantity** Can be used to represent specific quantity of the item

Item is involved in the following relationships:

- Item can contain any number of other items
- Item can have an item he is contained within
- Item can be part of a single ZoneInstance
- Item can be part of a single Floor

3.3 Technologies

This section provides an overview of the technologies chosen for the system's development and discusses the factors that influenced their selection, together with an explanation of why competing alternatives were not adopted.

3.3.1 Frontend Technologies

The main purpose of this layer is to provide all the functionality that enables the user to interact with the system. It represents the part of the system running directly in the user's browser and is responsible for rendering the pages, collecting user input, and if needed communicating with the backend to persist the data. The following are the selected technologies that implement this layer:

JavaScript JavaScript[1] is the standard programming language used in web development, as it is the language that browsers use. It is also possible to use TypeScript[2] for frontend development, which is a superset of JavaScript, as an alternative programming language. Although it has its advantages, like static typing, it was quite frustrating to use last time I used it, as most of the times libraries are made for usage with JavaScript. TypeScript then still requires using types for the library, although the library does not document the usage with TypeScript. Usually, the library has "headers" that make it possible to use together with TypeScript, but the lack of documentation really slows down development. That is why I have decided not to use it, even though it might be a better overall solution.

In retrospect, choosing TypeScript would have been a much better choice. The reason is that I have learned that there exists a non-strict mode for TypeScript that would allow us to use the TypeScript like JavaScript but with additional features like interfaces. The other reason is that the time saved using JavaScript was probably lost when refactoring the JavaScript code. Because, as the code base became larger and larger, refactoring the code is very dangerous. That is because as there are no compile time safety checks, JavaScript for example does not care, that function is used with incorrect number of arguments and the only way to check that the code was successfully refactored is by testing which in this application is not the easiest because of the high interactivity.

Node Package Manager Package Manager

The Node Package Manager[3] is the package manager used to install all frontend dependencies. It is the standard package manager that comes preinstalled with Node. Which is the most widely used JavaScript runtime.

Alternatives exist such as Yarn or pnpm. These alternatives advertise that they can download the packages faster or manage storage more efficiently, but even though they are nice things to have, I have decided to use npm as it is what I am most comfortable with.

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript
[2] https://www.typescriptlang.org/
[3] https://www.npmjs.com/

Phaser Phaser[4] is a 2D game framework for web applications. It will be used to create the editors in this application. As this application requires editors for the shapes/furniture/floors, it is very convenient to use a game framework to develop the editors as the framework comes with a lot of useful functionalities required to create them.

There are alternatives like Pixi.js[5], Three.js[6], or Babylon.js[7] but Three.js and Babylon.js are mainly used for 3D which this application currently does not require and if there is a requirement for the third dimension in the future, there is an extension for Phaser enable3d[8] that adds the possibility. Pixi.js was actually used at the start of development, but it was not sufficient since Pixi.js is mostly a rendering engine, so it felt like it was missing a lot of the functionalities required to implement the interactive functionality this application requires. Phaser actually uses Pixi.js under the hood, but comes with a wide array of useful interactive features on top of it.

Bootstrap Bootstrap[9] is a user interface (UI) library that contains some predefined UI components typically used in applications. Creating a user interface for a website that looks quite nice is not the easiest task. Bootstrap simplifies it as it has a lot of predefined nice looking components that can be used to create a nice looking website.

There are not many alternatives to Bootstrap. There are alternatives like Bulma[10], but Bootstrap is the most popular, which is why I have decided to use it. Other alternatives are usually used with some kind of JavaScript framework such as React[11], or Svelte[12]. Lastly, there is Tailwind[13] but I really do not count that as a good alternative because it does not have predefined components, so having to design them myself was a deal breaker as it is what I struggle with.

JSDoc JSDoc[14] is a documentation tool that uses comments in the JavaScript code. It reads the comment in the code and can generate documentation of the code. It makes it possible to write code and immediately document it without having to write a separate documentation for it. Additionally, it can be used as an alternative to TypeScript for type-checking, although it is not on the same level.

There are some alternatives such as ESDoc[15] or documentation.js[16], but the decision was made to use JSDoc because it seems to be the most popular.

[4] https://phaser.io/
[5] https://pixijs.com/
[6] https://threejs.org/
[7] https://www.babylonjs.com/
[8] https://enable3d.io/
[9] https://getbootstrap.com/
[10] https://bulma.io/
[11] https://react.dev/
[12] https://svelte.dev/
[13] https://tailwindcss.com/
[14] https://jsdoc.app/
[15] https://esdoc.org/
[16] https://documentation.js.org/

3.3.2 Backend Technologies

The main purpose of this layer is to provide the APIs responsible for processing the application's data by validating, transforming, and preparing it so that it can be safely and consistently persisted in the databases with which each service communicates.
The following are the selected technologies that implement this layer:

Java Java[17] is an object-oriented programming language that will be used to implement the backend of the application. The backend of applications can be programmed in basically any programming language out there. At the start of development of the application, I wanted to use PHP, but Java has a much more developed infrastructure and has a lot of libraries that are very useful for developing backend applications. Especially for the microservice architecture chosen for this project. A good example is the Service Discovery by Spring Netflix Eureka, which will be used in this application. It is quite difficult to find similar functionality for PHP.

Spring Boot Spring Boot[18] is an application framework for Java that will be used to create all backend services. It has modules for a wide range of requirements that a developer might need in his application. Using a framework such as Spring significantly simplifies the development of the backend microservices as it makes it really simple to define APIs and communication with databases, along with other functionalities.

There are other Java frameworks like Micronaut[19] and Quarkus[20]. Spring seems to have the most extensive community, and as a result it also supports the widest range of features.

Spring Cloud Gateway Spring Cloud Gateway[21] is an implementation of an API gateway for the Spring framework. It will gather all the backend APIs under one access point. It allows us to hide the location of the backend services from the client by having the client access the gateway and have the gateway send the request to the backend service. It also has some other functions like assigning some filters to the routes, limiting request rates, etc.

Spring Cloud Netflix Spring Cloud Netflix[22] is an implementation of a service discovery service for the Spring framework. It is used for discovery off all the backend services during runtime. it registers all the microservices used in the backend and the other microservices can then request the location of the other services based on a name. It makes it possible to change locations of the services, or add multiple of the same services without breaking the dependent service as it resolves the final location during runtime.

[17] https://www.w3schools.com/java/java_ref_reference.asp
[18] https://spring.io/projects/spring-boot
[19] https://micronaut.io/
[20] https://quarkus.io/
[21] https://spring.io/projects/spring-cloud-gateway
[22] https://spring.io/projects/spring-cloud-netflix

3.3.3 Database Technologies

The main purpose of this layer is to persist all the important application data so that the user does not lose its data.
The following are the selected technologies that implement this layer:

PostgreSQL PostgreSQL[23] server to serve as the main database to save all application data. It is an object-relational database management system. Although mainly a relational database, it supports managing more than just typical relational data. It supports creating custom data types, and what is unique is that it supports table inheritance. It also supports saving unstructured data as JSON. Furthermore, it has an extensive developer community that provides other features that might usually be provided by proprietary software .PostgreSQL also has good performance for small but also large applications. Additionally, it has good support for JSON data, so it should be possible to use it for all application data. Alternatives include MySQL,[24] and MariaDB.[25] The alternatives were not chosen, although they would do the job as well. PostgreSQL seems to have the most advanced features if there is a need in the future, and although it is not the most popular according to db-engines[26], when looking for which database to choose, PostgreSQL seemed to be the most recommended out of them.

pgAdmin PgAdmin[27] is a graphical user interface for accessing PostgreSQL. Although it is not a requirement for the application, it will be used for simplifying development, as it allows easy access to the saved data in the database and change values without having to write any structured query language (SQL), allowing quickly checking if we are saving data correctly.

3.3.4 Build and Deployment Tools

The main purpose of these tools is to provide the functionality required to build, package, and deploy both the frontend and backend components of the application. They ensure that the application can be prepared for production and reliably deployed across different environments.
The following are the selected tools:

Vite Vite[28] is a frontend build tool. It also contains a bunch of other features to simplify development such as Hot Reload, which automatically updates the contents of the website when some code change occurs. Having a development server hugely improves the development experience, and Vite works excellently when used inside a Docker container as well. It will also generate static assets for the frontend making it then possible to deploy the resulting frontend of the application anywhere.

[23] https://www.postgresql.org/
[24] https://www.mysql.com/
[25] https://mariadb.org/
[26] https://db-engines.com/en/ranking
[27] https://www.pgadmin.org/
[28] https://vite.dev/

There are alternatives such as Webpack[29], or Parcel[30]. The choice of Vite over Webpack is because Webpack is known to be quite difficult to configure while Vite works almost immediately out of the box. Parcel is also quite easy to configure, but Vite is currently quite more popular overall and seems to also build faster. In addition, many libraries include templates for use with Vite. The Phaser[31] library used in this project, for example, has an official Vite template, making it very easy to setup.

Maven Maven[32] is a build tool responsible for preparing the backend services to be ready for deployment. In addition, it is also used to manage the dependencies of the backend services. Any kind of build tool is a necessity, so it is possible to run the services at all.

The widely used alternative is Gradle[33]. Gradle is the newer tool of the two and is considered faster and highly customizable. Gradle, though, has a higher learning curve than Maven, and that is why I have decided to use Maven in this project.

Docker Docker[34] is a virtualization software used for easy deployment of the application and simplified development. It makes it possible to deliver software in packages called containers. These containers can be described using code in a Dockerfile. Anyone can run this container by just having Docker and is guaranteed to have exactly the same container eliminating the "It worked on my machine" phrase. It is also very convenient to deploy applications, as just one line of code is enough to deploy instead of having to install all the dependencies to make the application run.

Docker Compose can be used to run multiple containers, allowing one to additionally specify networks where the container should run. Docker Compose is very convenient to use with our microservice architecture, as every service can run in its own container. Furthermore, using networks allows us to expose only the API Gateway container to the user, completely preventing the user from directly accessing the databases.

A good alternative to Docker is Podman[35]. It is considered to be more secure as it is possible to run it without root access. It also does not require a daemon to run the containers. Docker, though, is considered the industry standard for containerization, having the widest support in a bunch of other tools. This significantly affected the choice of Docker.

3.4 Mockups

This section presents mockups of the application's pages, providing a visual representation of their expected appearance in the completed system.

[29] https://webpack.js.org/
[30] https://parceljs.org/
[31] https://phaser.io/
[32] https://maven.apache.org/
[33] https://gradle.org/
[34] https://www.docker.com/
[35] https://podman.io/

3.4.1 Item Location Mockup

The mockup illustrated in Figure 3.7 shows how the item location tracking screen might look. First, there is a header(number 1 in Figure 3.7), as is typical in most websites. In the upper left corner of the header, there is a logo(number 2 in Figure 3.7) of the application that can be clicked to return to the main page. Immediately next to the logo, there is a navigation bar(number 3 in Figure 3.7). In the mockup, there is currently only one button "Items", which leads to the inventory list. Depending on the role of the user, there can be more buttons in the navigation bar. As an alternative to the navigation bar, if there are many buttons, there can be a dropdown menu with the buttons. In the top center, there is a title(number 4 in Figure 3.7) with the name of the item being tracked. It should prevent the user from being confused if he has multiple tabs open to track multiple items at once. Then, as is typical in most applications, we get a button(number 5 in Figure 3.7) in the top right corner, which opens a menu with the user profile settings and logout button.

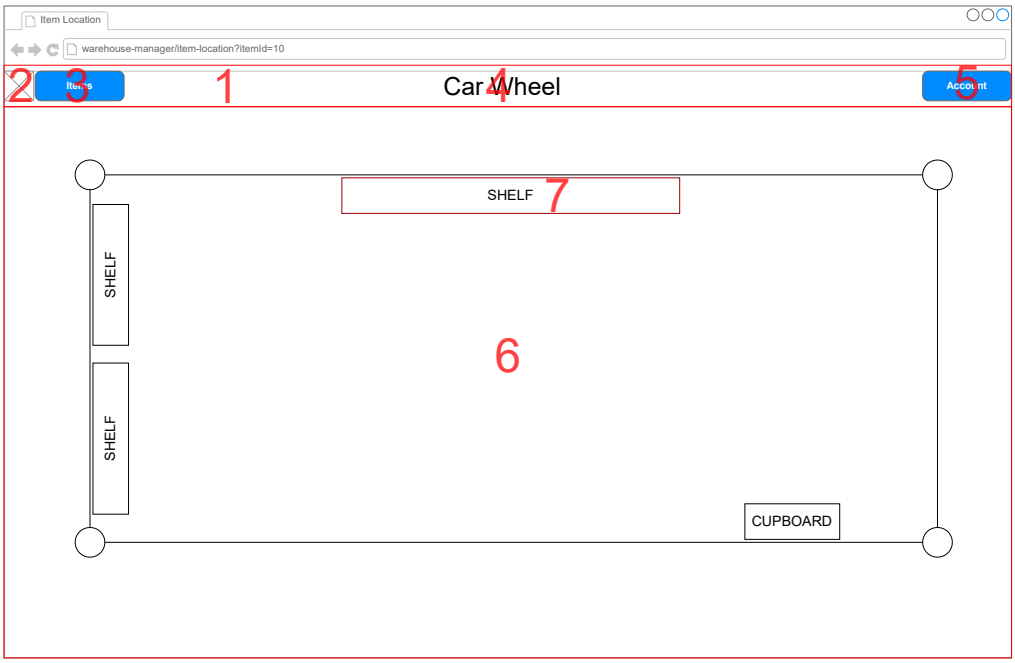


Figure 3.7 Mockup showing the item location page

This header is what I would consider proper in a production application. I would also consider switching from the navigation bar next to the logo to a drawer, which is currently more typical in most modern applications, as it is also more scalable if a lot of new menus need to be added. As there are no real users in the application, there will be no account button because there are no accounts to manage. Additionally, I will not focus on making proper navigation in the header, so focus is kept on the more important parts of the project. Instead, there will be a single main page with all the important navigation.

Below the header, there is a section(number 6 in Figure 3.7) with the item location. It shows the floor where the item is located and highlights in color the specific furniture(number 7 in Figure 3.7) where the item the user is tracking resides. The floor is shown as designed in the floor editor, with walls and furniture

placed on it. The highlighted furniture present in the floor should be clickable so that the user can see the interiors of the furniture where the user can see his placement zones. If the user clicks on the highlighted furniture, the zone with the searched item should also be highlighted. By pressing the zones, a modal window with the item trees should be shown. If the user presses the highlighted zone, the modal window should open and the path to the searched for item should be highlighted so that the user gets to see the exact item location to the specific box or other container.

3.4.2 Furniture Editor Mockup

This mockup illustrated in Figure 3.8 shows how the furniture editor might look. As in the previous mockup, there is a header(number 1 in Figure 3.8) with a logo(number 2 in Figure 3.8) in the top left corner, a navigation bar(number 3 in Figure 3.8) with buttons to navigate, and a button in the top right corner(number 4 in Figure 3.8) to open a menu with user profile settings and logout button. Again, the navigation bar may be substituted for a drawer for better scalability. Currently, in this project the navigation bar will be substituted by a single main page with all the navigation for this project, and the profile menu will be omitted as we will work in a single-user environment only. In the center of the header(number 5 in Figure 3.8), there is an input where the user will enter or edit the name of the furniture in the editor.

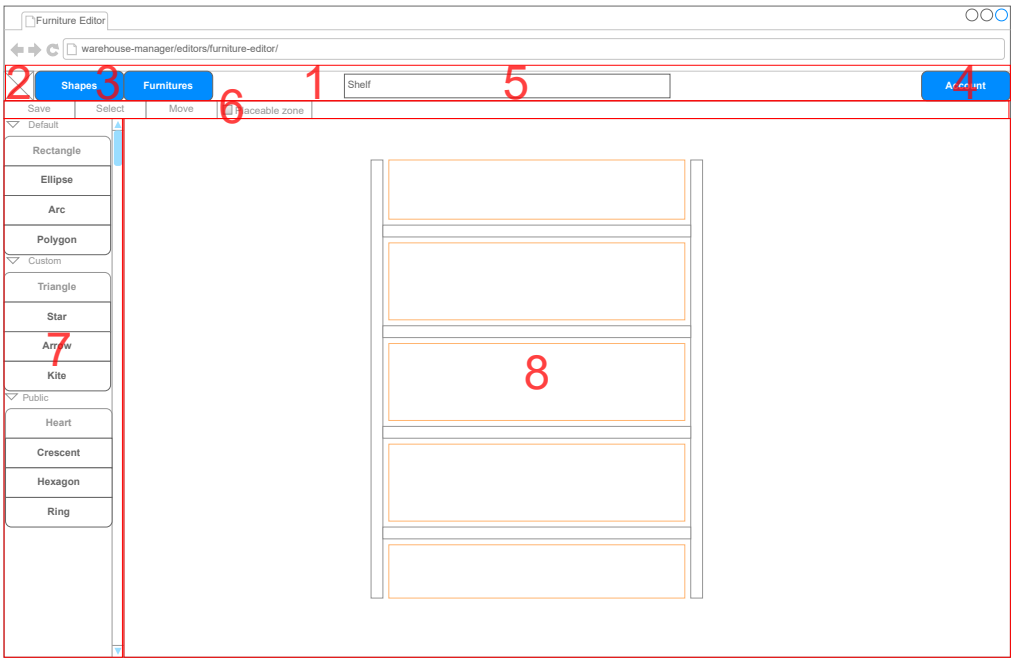


Figure 3.8 Mockup showing the furniture editor page

Below the header, the mockup shows a section(number 6 in Figure 3.8) intended for editor tools. In many applications, this is where tool buttons or tool groups are located. However, in this project, this section should be considered as a **future enhancement**. For the initial implementation, most editor actions will be performed through keyboard shortcuts or intuitive mouse interactions, minimizing UI complexity.

On the left side of the editor, there is a panel(number 7 in Figure 3.8) that is intended to list available shapes. In many furniture or design editors, shapes are either embedded directly in the toolbox or placed in a separate library panel. To avoid cluttering the tools section, the mockup places the shapes in a dedicated scrollable list. In addition, a search box for the list may be included to easily browse it. The list is separated into 3 sections: "Default", "Custom", and "Public". The "Default" shapes are the shapes that are provided directly by the system. The "Custom" shapes are the shapes created by the user or by other users of our organization. The "Public" shapes are the shapes that are shared by other organizations and can be used by the user to create furniture. These sections can be opened and closed by clicking on the section title, so the user can easily hide the sections he does not need at the moment. For simplicity, the shapes list is shown as a list of buttons with the shape name, but it might be better to use a grid layout with icons representing the shapes so that the user can easily recognize the shape he is looking for. For this version of the application the sections will be omitted, as there will be only the 4 system shapes, the shared section does not make sense in a single-user environment. That is the reason why the sectioning will be omitted for now.

The rest of the editor(number 8 in Figure 3.8) is the main area where the user can create the furniture. In the mockup, a simple shelf is shown. The construction of the furniture is shown using white lines, so the user can easily see how the furniture is constructed. In orange lines, the placeable zones are shown, so the user can easily see where the items can be placed within the furniture. The placeable zones are later used to place the items in the furniture.

3.4.3 Item List Mockup

This mockup illustrated in Figure 3.9 shows how the item list might look. As in the previous mockups, there is a header(number 1 in Figure 3.9) with a logo(number 2 in Figure 3.9) in the top left corner, a navigation bar(number 3 in Figure 3.9) with buttons to navigate, and a button(number 4 in Figure 3.9) in the top right corner to open a menu with user profile settings and logout button. But it will also be substituted by a single main page with all the navigation for this project.

Below the header, we have a section(number 5 in Figure 3.9) to control the list of items. Firstly, in the upper left corner, we have the title(number 6 in Figure 3.9) of the section, which is "Items" in this case. The reason is that the user can have multiple tabs, and if the user has permission to also view the other lists, he can easily distinguish which tab is which. Next to the title, there is a search bar(number 7 in Figure 3.9) that allows us to quickly search for a specific item in the list. These two elements are on the left side of the section. To make the section look better, the other elements are on the right side of the section. From the right side, the first element(number 8 in Figure 3.9) is a button to add a new item. This button may be disabled if the user does not have permission to add items. Next to the button, there is a button(number 9 in Figure 3.9) that opens a way to filter the items according to certain criteria. Lastly, there is a small dropdown menu(number 10 in Figure 3.9) to change the number of items shown on the page.

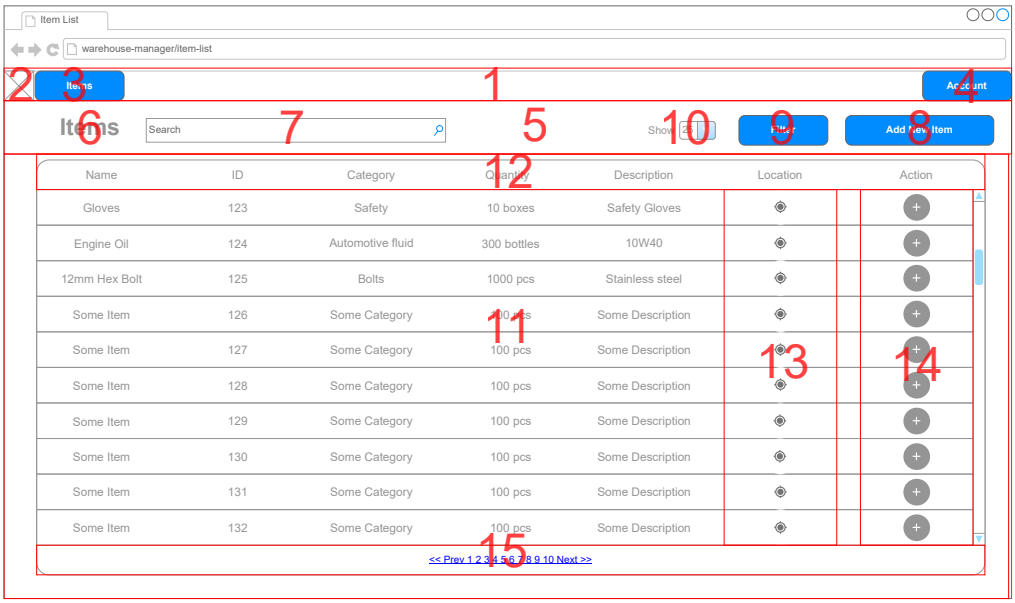


Figure 3.9 Mockup showing the item list page

Below the control section, there is a list of items(number 11 in Figure 3.9). Items are shown in a table with columns for item name, ID, category, quantity, description, location, and actions. The header of the table(number 12 in Figure 3.9) with the column names is fixed, so the user can easily see the column names even when scrolling through the list. Additionally, the column names might be clicked to sort according to those column names. Items are shown in rows, with each row representing an item. The location column(number 13 in Figure 3.9) contains a button that opens the item location tracking screen when clicked. The action column(number 14 in Figure 3.9) contains buttons to edit and remove the item. At the bottom of the list(number 15 in Figure 3.9), there is a pagination section to navigate the pages of items.

This way of displaying the items is very common in web applications, so users are used to it. It allows the user to easily find the item he is looking for and to perform actions on it.

4 Developer Documentation

This chapter describes the implementation of the application. It starts with a general summary of the implementation, followed by a description section describing interesting details about the implementation such as algorithms or data structures. It then explains the implementation of the user interface, followed by a description of the implementation of the backend. After that, it describes API to show how to extend the application. Then it explains some implementation problems that had to be solved during the implementation process. Finally, testing of the application is explained.

The complete source code of the application is publicly available at GitHub[1].

4.1 Implementation Summary

The application consists of a frontend and a backend. The frontend is responsible for all user interaction, visual editing, and presentation logic. The backend is responsible for handling persistence, versioning, and overall management of shapes, furniture, floors, and items.

The frontend is separated into multiple pages. There are pages for editors, views, item location, and element list pages. The editors are where the user creates the shapes, furniture, and floors. The views are screens where the user can use the created floor and furniture, and place his created items to the specific locations on the floor and within the furniture. The item location screen then highlights the specific location of the selected item. They are implemented using Phaser's Scene class which is Phaser's top level element after the Phaser `Game` class itself. Instead of the scenes having the logic implemented directly, functionality of the Scene is encapsulated into reusable classes called managers, which then add the functionality to the scene using composition. Each manager is responsible for a single concern, such as camera manipulation, shape movement, selection, undo/redo, or element creation. The scenes themselves are then used to configure these managers. Scenes are also responsible for initializing UI of the application. The user interface is initialized using classes that contain the logic. There are user interface classes responsible for initializing the list of elements that can be added to the editors, modals where parameters of the shape about to be added are set, or for initializing the save button. The logic of the floor, furniture, shape, and item list pages is implemented by using simple functions that are responsible for initializing the list, creating the pagination, or sorting.

The backend is implemented as a set of Spring microservices. Each management service is responsible for a specific domain entity, such as shapes, furniture, floor, or items, and exposes a representational state transfer (REST) interface. They are implemented simply where the request is intercepted by a controller which then uses a repository to communicate with the database to persist the data. Service discovery is handled using Eureka and an API gateway is responsible for routing requests to individual services.

Compared to the original design, the item list page does not have a proper

[1] https://github.com/notSnuffy/Warehouse-Manager

filtering button. Instead, it enables one to sort the items according to a given criterion. The reason for not implementing is accidental and should still be implemented in the future, as criteria sorting was implemented without realizing that it is not the same as filtering. Filtering is still possible, but only based on the item name.

Also, compared to what can be seen in the design documentation, although it was already mentioned there, the management services do not communicate with each other through the gateway. That is, to decrease the latency of the requests and not crowd the gateway. It is reserved only for public requests.

4.2 Algorithms and Data Structures

This section focuses on describing selected algorithms, data structures, and interesting facts that are used in the application.

4.2.1 Editor Scenes

The editor scenes are used for the creation of the shapes, furniture, and floors.

Shapes The scene for shapes is created by `ShapeEditor`. The scene is only responsible for configuring all the aforementioned managers as mentioned in Section 4.1. The only thing it implements is preloading an existing shape for update if requested. To be able to edit the existing shape, it is required to first unpack it from its bounding container, into which all the shapes in the editor are always packaged. In the future, it should be considered not to allow packaging of the shapes over and over again. There should be some check to not package the shapes into a container if there is only a single root shape in the editor overall.

Furniture The configuration of managers for the `FurnitureEditor` is much more extensive. The reason for this is that the shapes are separated into zones and the structural shapes. The `ShapeManager` for each then has to be configured individually, and because the zones work with labels, zones also have their own modal with an additional field for the labels. This highlights that the configuration of the scene may be too verbose and that an effort should be made to try to simplify it. The repeating code in the factory methods could be refactored into a predefined set of factory methods, or the camera managers could be packaged into a single preconfigured manager.

Floor The `FloorEditor` scene configures the managers, but also implements some own logic. The reason for this is the introduction of corners and walls. The walls are simple as they are only edges between the corners, although there is currently a problem that it is impossible to delete a wall without deleting a connected corner first, which is very inconvenient. But the realization that this feature is missing came too late in the development and there was not enough time to implement it. The problem lies in the corners. They highlight a flaw in the `SelectionManager` where the manager is unsuitable to use when selection is required, but the shape should not be resizable or rotatable. For that reason,

the `FloorEditor` implements its own logic for the selection of corners. Then, when there are two selected corners, a wall is placed. In addition, the editor implements its own logic for adding corners. That is because adding the corners is different from using the modals to add the shapes. For adding corners, the user first toggles a corner button, where within the editor, a preview of the corner is rendered before the user clicks a desired position in the editor to place the corner. Because of that, the `FloorEditor` has to manually add the commands for when a wall, or corner is added/deleted. The move operation is luckily simple as the `MoveManager` can be reused but the editor has to check if the shape moved was a corner and in that case it has to move its connected walls. For quick checking of the neighbors of a corner, an adjacency list is currently saved in the `FloorEditor`. Although this helps to increase responsiveness, it unnecessarily wastes memory. Saving neighbors directly in the corners would save some of the memory while keeping the same logic. In this case, though, a viable option would be to not keep track of the neighbors at all and manually finding them by traversing all the saved walls in the associated `ShapeManager`. The reason is, that in the real world, it is unusual to find floors, or rooms that would contain a huge amount of walls and traversing under a thousand of elements is efficient enough to maybe justify not keeping track of the neighbors separately.

4.2.2 View Scenes

The views are scenes where the user can see a view of his floor, or room and press the placed furniture to see a view of the placed furniture. The user can subsequently place his defined items in the zones defined in the furniture. These views where initially not planned to be implemented. The initial expectation for the movement of items was that they would have something similar to a multiselect transfer as shown in Figure 4.1[2]. But the decision was made that it would not be very user-friendly, so this option was chosen as it should be overall more intuitive for the user compared to the alternatives.

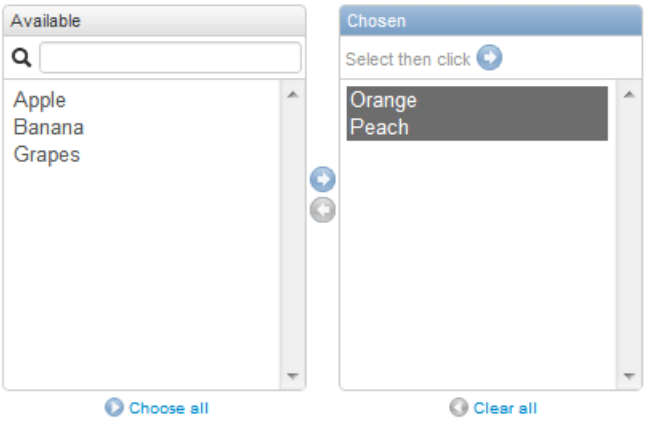


Figure 4.1 Multiselect Transfer Example

The `FloorView` scene acts as the main scene, and the `FurnitureView` scene is the subscene. There are currently 2 different options to switch from the `FloorView`

[2] https://stackoverflow.com/questions/11165884/multiselect-two-column-transfer-widget-with-twitter-bootstrap-theme

scene to the `FurnitureView` scene. On each placed furniture, a press handler is implemented that switches the scenes, or the user can drag an item from the list on the left for some given time, which then switches the scene. To get back to the `FloorView` scene, the user presses the `Back` button in the `FurnitureView` scene.

First, the switching between scenes will be explained. The `FloorView` scene because it serves as the main scene should never be destroyed and the state should not reset. For that reason, when switching the scene, it goes to sleep, and when it comes back, it wakes up. The `FurnitureView` scene is a bit more complicated. It should be completely reset when we go to the `FurnitureView` scene, because it must be able to render all the different furniture that is placed. It is done by starting the scene when entered and stopping when exited. It forces the scene to start from the start. Although the scene starts from the beginning, there was a problem that it did not completely reset. More about the problem and its solution can be found in Section 4.6.6.

The `FloorView` scene is responsible for saving all the floor data. This ensures that all data are always up to date. If the floor did not have all the data in memory and loaded the data for the furniture instance when going into the `FurnitureView` scene, the items in this furniture would not be up to date if the user did not save his changes. For that reason, the furniture instance is always passed as a reference in the `FurnitureView` scene. The `FloorView` scene implements a method called `moveItemBetweenZones`. This method can be seen in the Listing 4.1. The method is responsible for the movement of items between zones that are present in this floor. It ensures that if other furniture instance is rendered, it has all the up to date data. The method currently tries to find the zone by searching for it linearly. It may require some optimizations in the future, but currently the solution is good enough, because it would be unusual to have thousands of placable furniture and additionally it would be unusual to have thousands of zones in the furniture. This means that in the case of normal usage, no performance issues should be observed.

Listing 4.1 moveItemBetweenZones method

```
moveItemBetweenZones(previousZoneId, newZoneId, itemId) {
    this.#furnitureInstances.forEach((furnitureInstance) => {
      furnitureInstance.zoneInstances.forEach((zoneInstance) => {
        if (zoneInstance.id === previousZoneId) {
          if (!zoneInstance.items.has(itemId)) {
            console.warn(`Item ${itemId} not found in zone `
            + `${previousZoneId}`);
            return;
          }
          zoneInstance.items.delete(itemId);
        }
        if (zoneInstance.id === newZoneId) {
          zoneInstance.items.add(itemId);
        }
      });
    });
}
```

In addition to saving all the floor data, it also saves all the items. It gets all the items when populating the item list. From these items, it does not save all the fetched item data. It saves only the name that is used to render the item for distinction, the location of the item(floor identifier, zone identifier, and parent

identifier), the identifiers of the children, and lastly it keeps a flag if some change for this item happened. It can be seen in the Listing 4.2. In this case, we save the items inside a map for constant access because there might be many items the users have created. Saving all of the data at the same time may become a problem in the future, as it may become impossible to keep all of the items in the memory at the same time. For that a support in the API should be introduced to be able to fetch the items in batches, and the item map should be build up lazily so it includes only the items that are actually needed. The item list is currently only a simple list that lists all the items represented by their names. In the future, the list should get some enhancements so it is easier to use. It should have a search bar so it is easier to find the specified item, some filtering should be implemented so the user can, for example, filter based on the category. It is also inconvenient that currently it is impossible to tell if an item is contained within another item or if it contains some items itself. The reason is that dragging an item to a zone automatically takes it out of its parent and additionally takes all the item's children with it, making it currently hard to tell what moves without knowing it prior.

Listing 4.2 Item Entry in the Item Map

```
let children = new Set();
item.children.forEach((child) => {
    children.add(child.id);
});
const itemEntry = {
    name: item.name,
    floorId: item.floorId,
    zoneId: item.zoneId,
    parentId: item.parentId,
    changed: false,
    children: children,
  }
```

Next, the event handler responsible for the hover event over the objects will be explained. This hover functionality is impossible to implement using Phaser's functionality alone as the items are dragged from outside Phaser. The problem and solution are explained in Section 4.6.7.

After a hoverable object is detected, a delayed functional call is then created for half a second that then switches the scene, or in the `FurnitureView` opens a model with the zone's items. Additionally, checks are made that the user still hovers over the same object, otherwise the delayed call is removed. The Listing 4.3 shows the handler excluding some of the checks that prematurely exit the handler.

Listing 4.3 Canvas Hover Handler

```
#handleCanvasHover(e) {
    const gameCanvas = this.game.canvas;
    const canvasBdRect = gameCanvas.getBoundingClientRect();

    const scaleX = this.scale.displayScale.x;
    const scaleY = this.scale.displayScale.y;

    const canvasPointerX = (e.clientX - canvasBdRect.x) * scaleX;
    const canvasPointerY = (e.clientY - canvasBdRect.y) * scaleY;
```

```
    const pointer = this.input.activePointer;
    pointer.x = canvasPointerX;
    pointer.y = canvasPointerY;

    const hits = this.input.hitTestPointer(pointer);

    if (hits.length === 0) {
      this.#clearHoverTimer();
      return;
    }

    const topHit = hits[0];
    if (this.#lastHover && this.#lastHover !== topHit) {
      this.#clearHoverTimer();
    }

    if (this.#hoverTimer) {
      return;
    }

    this.#lastHover = topHit;
    this.#hoverTimer = this.time.delayedCall(500, () => {
      this.scene.sleep();
      this.scene.launch("FurnitureView", {
        furnitureInstance: this.#furnitureInstances.get(
          topHit.metadata.furnitureInstanceId,
        ),
      });
      this.scene.bringToTop("FurnitureView");
      this.#clearHoverTimer();
    });
}
```

The `FurnitureView` is currently more of a prototype. The code is written so that it is as easy as possible to implement the functionality, but the code desperately requires a refactor. As an example, the commands intended for the `UndoRedoManager` are currently created as raw JavaScript objects rather than being represented as a separate class. There is a lot of code duplication between the creation of the main zone list and the children lists. Because there are some small similarities between their implementation, such as a child item should have its zone identifier set to null to indicate that it is not a root item, but some centralized creation of the lists would be preferred.

To simplify the implementation of dragging items between lists, the Sortable[3] library was introduced. This made the implementation of dragging the HTML elements between the lists very simple. Using the `onAdd` method that Sortable implements on the lists, it is also quite straightforward to modify the values in the item map that represent the changes and move the items between zones. There were problems getting the undo/redo functionality to work for the item movement. The problems are explained in more detail in Section 4.6.8 and the solutions described.

There is currently an issue with the undo/redo system, where to indicate if an item was changed, it has a flag. The problem is that this flag cannot be safely changed to the previous state without issues. The issue is that if the user

[3] https://github.com/SortableJS/Sortable

performs an item action, the flag is set to indicate that the item was changed. If the user now saves this to the database and subsequently does an undo, if the flag is changed to its previous state, the item would not be saved again. This at points leads to a duplication of rows in the database, but the user gets a consistent experience.

4.2.3 Item Location Scene

The scene for the location of the items is basically the read-only alternative to the view scenes described in Section 4.2.2. The difference is that, while loading the floor data, it additionally flags specific furniture, zone, and item. To find the items that should be highlighted, it uses the depth-first search algorithm (DFS) which can be seen in the Listing 4.4. It then flags the path that it should be highlighted, and upon rendering it colors every flagged element in red to highlight it.

Listing 4.4 Find Item Algorithm

```
const findItem = (item, path = []) => {
    if (item.id === highlightedItem.id) {
      console.log("Identified item to highlight:", item);
      return { item, path };
    }

    if (!item.children || item.children.length === 0) {
      return null;
    }

    for (const childItem of item.children) {
      const foundItem = findItem(childItem, [...path, item]);
      if (foundItem) {
        return foundItem;
      }
    }
    return null;
};
```

4.2.4 Undo/Redo

The `UndoRedoManager` is responsible for all the undo/redo operations of the application. Internally, it maintains two stacks (undo and redo) which store executed commands. The maximum size of these stacks can be configured, allowing us to limit the amount of memory used to store the history, as the oldest commands are discarded when the size is exceeded.

The manager exposes a simple interface consisting mainly of `undo` and `redo` methods. In addition, it allows one to clear the entire history, query the sizes of both stacks, and check whether undo or redo operations are currently possible. These checks are used by the user interface to disable actions when no corresponding operation can be performed.

The commands pushed to the manager follow the command pattern. More complex actions are represented using the composite pattern. For example, when a labeled shape is moved, a `CompositeCommand` is created that contains both

a `MoveShapeCommand` and a `MoveLabelCommand`. This ensures that undoing or redoing the action consistently restores both the shape and its label.

The manager itself acts as a publisher in the observer pattern, allowing observers to listen to its events.

4.2.5 Shape Types

This section describes the implementation of shape object classes. There are five classes for all the basic shapes. They all extend from their Phaser counterpart, adding additional functionality to the classes. The `Arc`, `Rectangle`, `Ellipse`, and `Polygon` classes represent the most basic shapes currently in use. They implement only a single additional method compared to the original Phaser class. They implement a `createSnapshot` method. The reason why it is implemented in these classes is based on the prototype pattern, although our method does not create a new instance of the class, just a snapshot. The container class additionally implements methods to calculate its edge points, such as top left and top center. Phaser does not implement them, but they are used to calculate bounding boxes, to create the resize handles, etc.

The `createSnapshot` method currently uses two special methods to calculate the real coordinates and dimensions of the child shapes. These functions are shown in Listing 4.5.

Listing 4.5 Methods to get position and size

```
function getRealDimensions(shape) {
  const width = Phaser.Math.Distance.BetweenPoints(
    shape.getLeftCenter(undefined, true),
    shape.getRightCenter(undefined, true),
  );
  const height = Phaser.Math.Distance.BetweenPoints(
    shape.getTopCenter(undefined, true),
    shape.getBottomCenter(undefined, true),
  );
  return { width, height };
}

function getRealPosition(child, parent) {
  const parentWorldMatrix = parent.getWorldTransformMatrix();
  const childWorldMatrix = child.getWorldTransformMatrix();

  const dx = childWorldMatrix.tx - parentWorldMatrix.tx;
  const dy = childWorldMatrix.ty - parentWorldMatrix.ty;

  const rotation = parentWorldMatrix.rotationNormalized;

  const rotatedX = dx * Math.cos(-rotation)
    - dy * Math.sin(-rotation);
  const rotatedY = dx * Math.sin(-rotation)
    + dy * Math.cos(-rotation);
  return { x: rotatedX, y: rotatedY };
}
```

The function itself works by first calculating the relative position of the child in world space by subtracting the world coordinates of the parent and the child.

Since this relative position still includes the parent's rotation, the rotation is undone to get the child's local position within the parent's coordinate system.

4.2.6 Adding Shapes to the Scenes

The `ShapeManager` was not implemented at the start. Initially, the added shapes were simply collected in a collection, and there was a function that created them from the list of elements 4.3.1. The problem was that it was poorly extensible and it also did not support other features which could possibly be implemented using the old code, but it would not be convenient.

To construct the manager, it accepts the scene the manager is associated with and it can also accept an object where key is some id of a wanted manager and value is the manager instance. The ShapeManager uses it to register a created shape with the given manager if it contains the manager id. It is usually used in conjunction with the managers that are responsible for manipulating the shapes, such as move, rotation, etc.

The `ShapeManager` uses the Registry pattern to register all the shapes that the manager should allow to create. More about registering new shapes can be found in Section 4.5.6.

Adding shapes to the manager is made mainly by using the `addShape` method. There are also `addShapeFromSnapshot` and `addShapeWithCommand` methods, but these both methods call the `addShape` method in the end. The `addShapeFromSnapshot` method creates a new instance from a snapshot of a shape, and the `addShapeWithCommand` method is used to add a shape normally, but afterwards also returns an object based on the Command pattern that can be used to undo the operation. It is used together with `UndoRedoManager` 4.2.4. There is also an `addExistingShape` that takes an existing instance of a shape and makes it managed by this manager, although it does not support reassigning from other managers. To add a new shape, the shape type should always be specified first. Afterwards, the parameters that are supposed to be passed to the factory function are specified while registering the shape type. Then it allows specifying some additional data. An interactive object can be specified which overrides the default configuration, some additional metadata can be specified (e.g. the id of the instance within the database) and the identifiers of the managers with which the shape should be registered. The identifiers should be one of the specified during the construction of the manager. Lastly, it should be specified whether adding the shape should emit an event or if it should be added quietly. This is required because adding a child element of a container should not then be intercepted, for example, by the `CameraBoundsManager` explained in Section 4.2.13, as the parent is already registered and the bounds include the child.

The manager itself does not construct the instance of the shape. For that, it uses the `ShapeFactory` that understands how to combine all the passed arguments and construct the instance using the factory.

Removing shapes from the manager is done using the `removeShape` or `removeShapeWithCommand` methods. They accept the reference to the shape that should be removed and additionally to the event if it should be emitted or be quietly removed.

There are general methods to check if the manager contains a given shape or

to get all the shapes in the manager. The interesting thing here is that there are two methods to get all shapes. There is a method to get all shapes in general and a method to get only the root shapes. That is because we also save children of containers in the collection of all the shapes, although they are technically already saved inside the parent. That is, to prepare the interface for the future where some unlocking mechanism might be implemented for the containers, and so it would allow manipulating the children of the containers individually while currently it is possible to manipulate only the root container. In that case, having access in constant time to the child would be preferred.

The `ShapeManager` currently uses the references to the shapes as the keys to its map. That was not always the case and more about it is explained in Section 4.6.5. That section also explains that the removal of the shapes had to be changed to soft-deleting for the approach using references to work correctly. The soft-deletion works well, but it is important to note that it can currently lead to memory leaks. The problem is that the JavaScript garbage collector will never collect the shape, because it is always saved in the manager. That is a problem, when nobody outside the manager holds reference to this shape as that means there is nobody that could ever restore the shape causing the memory leak. This can, for example, happen if all the commands in the `UndoRedoManager` 4.2.4 that held the reference to the shape where in the redo stack which was cleared as a consequence of adding a new command to the `UndoRedoManager` 4.2.4 which clears the redo stack. For that some solution, such as reference counting, should be implemented to get rid of this issue.

4.2.7 Adding Label to Shapes

The `ShapeLabeler` is a manager responsible for creating labels for given shapes. Creates a given label text in the center of the specified shape. Optionally, if specified upon the construction of the manager, the label can be modified by pressing it. For that reason, when adding a new label, the shape to which the label should be added must be specified, and the text and color of the label should also be specified. If the labeler allows for editing, a callback method should be passed, which is called after the update takes place. The callback can, for example, update the shape's metadata where the name of the shape is saved.

The manager also has an add method that creates the label from a snapshot for undo/redo functionality.

Additionally, there is a method to remove the label or get a specific label based on the shape.

The `ShapeLabeler` class can also work as a publisher in the Observer pattern. The reason for that is that the events that this manager creates are not really necessary for other managers. So instead of crowding the scene with events. If an event from this manager is required (specifically it is just a `labelTextChanged` event), then the observer should subscribe directly. In reality, this event is used only in the `LabeledCreateCommandEventHandler`, which is responsible for intercepting the event and creating a command to place in the `UndoRedoManager` 4.2.4 for the undo/redo functionality. The reason for having a separate class to intercept the event is that it should not be the manager's responsibility to know if undo/redo is active. In this way, the manager is much more portable.

4.2.8 Save/Load Shapes

This section describes how the serialization and deserialization of the shapes into instructions happen.

The class responsible for that is the `ShapeInstructionsHandler`. Because, as mentioned in 4.5.6 section, `ShapeManager` has some settings that specify the commands. For that reason, the `ShapeInstructionHandler` depends on a specific `ShapeManager`.

The `ShapeInstructionsHandler` has only two important methods. One to convert into instructions and one to convert from instructions. It works with snapshots of the shapes instead of using instances directly. It allows for more flexibility, as creation of the instances can be delayed to later time, and additionally the snapshots are a bit more convenient to work with rather than the shape itself. Additionally, it guarantees that the class does not make any unwanted modifications to the shapes. Since some elements can act as a container, stack is used to recreate the shapes from instructions.

4.2.9 Moving Shapes

The functionality of moving shapes by dragging is implemented using the `MoveManager`. Implementing dragging using Phaser is simple as it directly introduces drag events for its game objects.

To make it possible for the user to see the edges of the dragged shape. An `OutlineManager` is introduced that draws a specific outline. The `MoveManager` currently forces using rectangular dotted outline, but the `OutlineManager` itself accepts in its `create` method for the shape a function that accepts the graphics object to which the outline should be drawn and additionally the shape to which the outline should be added. This makes it possible in the future to create different outlines in the `MoveManager` based on the shape's hit area, rather than forcing the rectangle.

This manager also implements an `update` method, which might seem a bit unusual at first, since the `create` method creates event listeners on the shape, so it automatically updates. The reason for the update method is to be able to force the dragged shape to move into a different position. As an example, if the user is dragging a shape and at the same time panning the camera by dragging the shape on the edge of the screen. No drag event gets triggered as the user's pointer is stationary although the camera is panning. For that, in the scene's `update` method, we check if the camera is currently panning and a shape is being dragged, and in that case we use the `update` method of the manager to force the shape to move.

4.2.10 Rotating Shapes

The RotationManager is responsible for adding functionality for rotating shapes . It has three methods: `create`, `update`, `hide`. `Create` is responsible for creating the rotation knob that the user can drag to rotate the shape. `Hide` is responsible for hiding the rotation knob, and lastly `update` is used to update the location of the rotation knob in case the position of the shape changes or the shape is resized so it is always above the shape. Only care had to be taken in that in Phaser zero

radian rotation is on the right side of the shape, so adjustment has to be made to keep the knob above the center.

It also has a method `addManagerToUpdate`, to which an other manager can be passed, and if something forced the `RotationManager`, it also forces an update of the other manager. Mostly used in conjunction with the `ResizeManager` 4.2.11.

4.2.11 Resizing Shapes

The `ResizeManager` is responsible for implementing resizing for shapes and was the most problematic to implement, as resizing a shape is not as simple as just changing its size. Eight handles are created. One for each corner and one for each center of an edge. For each of these handles, a separate class is implemented where each implements an `accept` method that accepts a visitor. Compared to statically typed languages, JavaScript does not support parameter-based function overloading. So, the `accept` method of the handle must correctly call the method of the visitor intended for itself. The visitor pattern is used here, so long condition statements do not have to be used that check which of the handles is currently being dragged. Additionally, the functionality that visitors implement is not something that should be the responsibility of the handles. There are two visitors implemented: `MoveDimensionVisitor` and `MoveHandlerVisitor`. `MoveHandlerVisitor` is used to calculate the new coordinates and dimensions of the shape. `MoveDimensionVisitor` calculates the new dimensions of the shape.

The calculation is performed in this way. First, we always take the opposite handle to the one being dragged which is taken as the static point which we know should not move when resizing. The pointer position is than the new expected point, the dragged handle should end that. In case of dragging one of the center handles, we create a line from the static point through the old position of the dragged handle and find the nearest point on that line from the pointer position. To get the new coordinates of the shape, we calculate the center point between these two points as we add shapes with the origin at the center of the shape. To then calculate the new dimensions, we first undo the rotation of the points, so we get the points as if there were no rotation. That makes it easy to calculate the new dimensions. We just have to make sure the dimensions do not go to the negatives, and for the center handles, we always fix one of the dimensions so that it does not change. The pseudocode for this calculation can be seen in Algorithm 1.

Currently, the prevention of going into the negatives correctly stops at the given limit if the user moves the mouse smoothly. If the mouse is moved around violently and the mouse goes to the negative dimensions too quickly, the shape dimension can stop at a much higher size.

It also has a method `addManagerToUpdate`, to which an other manager can be passed, and if something forced the `ResizeManager`, it also forces an update of the other manager. Mostly used in conjunction with the `RotationManager` 4.2.10.

4.2.12 Selecting Shapes

The `SelectShapeManager` is responsible for selecting a specific shape. Currently, there is no manager that could select multiple shapes at the same time, but it is thought of as a good future enhancement.

Algorithm 1: Resize Shape Algorithm

```
Input: draggedHandle – handle being dragged
       pointerPosition – current pointer position
       shape – the shape being resized
staticHandle = GetOppositeHandle(draggedHandle)
staticPoint = GetPosition(staticHandle)
oldDraggedPoint = GetPosition(draggedHandle)
newDraggedPoint = pointerPosition
if IsCenterHandle(draggedHandle) then
    resizeLine = LineThrough(staticPoint, oldDraggedPoint)
    newDraggedPoint = GetNearestPointOnLine(resizeLine, pointerPosition)
end
newCenter = GetCenterPoint(staticPoint, newDraggedPoint)
unrotatedStaticPoint = RotateAround(staticPoint, newCenter, −shape.rotation)
unrotatedDraggedPoint = RotateAround(newDraggedPoint, newCenter,
  −shape.rotation)
newWidth = |unrotatedDraggedPoint.x − unrotatedStaticPoint.x|
newHeight = |unrotatedDraggedPoint.y − unrotatedStaticPoint.y|
if IsHorizontalCenterHandle(draggedHandle) then
    newHeight = shape.height
end
if IsVerticalCenterHandle(draggedHandle) then
    newWidth = shape.width
end
newWidth = GetMax(newWidth, MAX_WIDTH)
newHeight = GetMax(newHeight, MAX_HEIGHT)
shape.center = newCenter
shape.width = newWidth
shape.height = newHeight
return shape
```

The `SelectShapeManager` is made up of the `RotationManager` 4.2.10 and `ResizeManager` 4.2.11. It is responsible for activating the handles of the managers. Most of its functionality is to activate and deactivate the handles. For example, it hides the handles when the shape is being dragged, so that the handles do not interfere with the outlines. Additionally, it requests a shape removal when the shape is selected, and the user presses the **DELETE** key.

This manager also forces update for the `ResizeManager` 4.2.11 if the shape is resized and the camera is panned similarly as already explained in the Move section 4.2.9.

4.2.13 Infinite Editor World

The infinite world was implemented as a consequence of the problems mentioned in Section 4.6.3. For the editor to feel adequate, it felt necessary to implement all the functionality mentioned later in this section, to bring the editor closer to other existing editors. These editors have some common features that all users expect, so not including them would make the editor basically unusable.

The infinite world means that the user can place elements in any coordinates he wants.

To make the infinite world work correctly, it is required for the user to be able to move the camera so that it is possible to access elements outside the current

camera view. Responsible for the camera panning is the `PanningManager`. It has only two important methods. Method `create` that is responsible for registering event listeners on the scene that implement the panning. Additionally, there is an `update` that should be called in the scene's update method if the camera should be panned when the left mouse button is pressed on the edge of the screen. This can be used to enable the camera to pan when the user is currently dragging or resizing an element. The manager uses the associated scene as a mediator to inform other managers that want to observe that the camera was panned, reducing the dependency between the managers.

The `ScrollbarManager` also enables panning the camera by initializing scrollbars that pan the camera. For the scrollbars to work correctly with the use of dynamic bounds which are explained later in this section, the manager has to accept a list of event names it should observe on the scene for it to trigger synchronization of the scrollbars. The synchronization is responsible for resizing the possible range the scrollbars can move, as otherwise the scrollbars would pan the camera only in fixed dimensions, and it also has to synchronize the current position if the camera was panned by other means, so the user does not get randomly transported to seemingly random locations.

The scrollbars are implemented using the HTML input range element. Whose styling has a lot of quirks. Most browsers have their own specific options for styling the element, and although browser-agnostic options were used, it does not guarantee it will work as expected in all the browsers. It was tested using Firefox[4] and Chrome[5] , where the behavior seems to work as expected.

Although it should be implemented in the other managers in the future, the `ScrollbarManager` is one of the only managers that implements a `destroy` method. The reason for that is that for the view scenes, the scrollbars are reused for both of the scenes that are switched around. So, it is required to unregister the event listeners on the scrollbars before switching the scene, or it tries to use the scrollbars on the scene that does not exist anymore.

The main motivation for having camera bounds, although the user is allowed to place elements wherever he wants within the editor, is to have the camera more focused on the user's working space, as it does not allow the camera to move beyond the bounds by panning the camera. Additionally, it helps with synchronizing the scrollbars used for panning the camera if the camera is panned by other means, because the scrollbars cannot be infinitely long. This way if the camera hits the bounds, the scrollbar is at its edge position. The camera bounds are dynamically extended if any object falls outside the bounds by the `CameraBoundManager`.

At first, the `CameraBoundManager` was implemented with being also able to shrink the camera, trying to keep the camera being the smallest bounding rectangle. Although the functionality still works, it is not recommended to use as it makes the user experience quite unpleasant. It is better to keep the camera in its extended state. The manager can be configured by setting the events that it should observe on the scene. All the events this manager accepts should in its context contain the shape the event triggered for, so the manager can try to update the bounds using the shape's updated coordinates. For each event, it accepts whether it should

[4] https://www.firefox.com/en-GB/
[5] https://www.google.com/intl/en_US/chrome/

shrink, and additionally a padding in pixels that can trigger the adjustment. The padding is required for some events where the edge of the shape is used instead of the origin, such as resizing. Otherwise, the bounds will not extend.

For finding the edge coordinates, a `SortedLinkedMapList` is implemented. The `CameraBoundsManager` then contains four of these lists for each side. It is a linked list kept in sorted order, where the head always contains the edge coordinate. For that, the class accepts a comparator function to determine the order of the elements. To make the list more efficient, the list is supported by a hash map. This map contains all nodes of the linked list. For that reason, to insert into the list, key-value pair has to be provided. The manager uses references to the shapes as keys and the coordinates as the values. The map is then used to check in constant time if the expensive reinsertion should happen after updating the value for a specific key by checking if the value upon comparison still belongs between the previous and the next node.

To also make the editor more convenient to use, zooming is implemented. Implementing zooming is not as straightforward as it would seem as just zooming the camera zooms towards the center, which is not what is usually expected by the user. The user usually expects that the zoom will be towards his pointer. To calculate how much the position of the camera should be compensated, it uses the pointers relative position within the screen to calculate the adjustment. The whole algorithm can be seen in the Listing 4.6

Listing 4.6 Zoom Towards Pointer Algorithm

```
const oldZoom = camera.zoom;

const zoomDirection = deltaY > 0 ? -1 : 1;
let newZoom = camera.zoom + zoomDirection * this.#zoomStep;
newZoom = Phaser.Math.Clamp(newZoom, 0.5, 4);

const cameraPointerPosition = {
  x: pointer.x,
  y: pointer.y,
};

const cameraCenter = {
  x: camera.centerX,
  y: camera.centerY,
};

const pointerOffsetFromCenter = {
  x: cameraPointerPosition.x - cameraCenter.x,
  y: cameraPointerPosition.y - cameraCenter.y,
};

const ptrOffsetFromCenterRatio = {
  x: pointerOffsetFromCenter.x / camera.width,
  y: pointerOffsetFromCenter.y / camera.height,
};

const pixelDiffAfterZoom = {
  width: camera.width / oldZoom - camera.width / newZoom,
  height: camera.height / oldZoom - camera.height / newZoom,
};
```

```
const scrollAdjustment = {
  x: pixelDiffAfterZoom.width * ptrOffsetFromCenterRatio.x,
  y: pixelDiffAfterZoom.height * ptrOffsetFromCenterRatio.y,
};

camera.setZoom(newZoom);
camera.scrollX += scrollAdjustment.x;
camera.scrollY += scrollAdjustment.y;
```

4.3 User Interface

This section explains how the user interface of the application is created.

As no JavaScript framework is used, most of the HTML is created statically using HTML files. That HTML is then accessed using JavaScript to add the event listeners that trigger the elements specific functionality. There are some exceptions where the HTML is rendered dynamically, which will be mentioned later in this section.

The initialization of the user interface occurs in the scene if it is present. That is, for simplicity, as in that case, the user interface modifies the scene. However, it is quite late to initialize it there. For example, the user can observe that the shapes in the shape list are not yet present. It should definitely be looked at in the future to make the experience feel smoother.

4.3.1 Element Lists in Editors

The element lists are responsible for allowing the user to add elements to the editor. Note that only the shape list will be explained, as the furniture list works the same with only small differences, such as different data attribute, or starting without any system furniture. Unfortunately, the code currently does not reflect that, as the furniture list does not inherit from the shape list and is currently its own whole implementation. In the future, some generalization of the classes would be in order.

Responsible for initializing the list of shapes is the `ShapeListUserInterface`. The class accepts the identifier of the HTML element acting as the list of shapes, a function to be called when the shape in the list is pressed, and the URL of the shape management API so that it can fetch the saved shapes.

To initialize the list, the `initialize` method must be called. That is, in case we would not want to initialize the list with the data from the database and instead wanted to manage the list manually using its `addShapeIntoList` method.

Currently, it has to be manually ensured that the identifiers of the system shapes in the database and the system shapes of the editor are the same. It is made by using the `ShapeTypes` enumeration which would have to be changed in case the identifier in the database changed. In the future, it should automatically try to get the system shapes in the database and use the identifiers from there.

Listing 4.7 ShapeTypes enumaration

```
const ShapeTypes = Object.freeze({
  RECTANGLE: 1,
  ELLIPSE: 2,
```

```
  ARC: 3,
  POLYGON: 4,
});
```

4.3.2 Element Modals in Editors

This section explains the modals that open when the user presses an element in a given list 4.3.1. As already mentioned in Section 4.6.3, the modals are remnants of the old way to add the elements to the editor. Changing the world to have unlimited size makes the modals inconvenient to use since if the user currently works in different parts of the world than the origin, it is impossible to tell to which coordinates should be the new element added. It then becomes a guessing game where the user basically has to guess the coordinates and if he does not see the element, he has to undo the action and try again, or he has to place the element to origin and drag it to his working area. Because the fields in the modal are rendered dynamically, it is also not straightforward to try to change the field to be the center of the camera. Even though changing the value of the field after the modal opens would work, it would be more of a temporary fix, as using the modal makes it tedious for the user overall. It requires the user to do an additional click to add the element to the editor. A better way to add the elements would be to always add the element with default values to the editor in the center of the camera. Then, there would be a menu in the editor where the user can change the properties of the element that is selected. This is how it is usually implemented in other editors. An example of an editor that handles the adding in such a way is draw.io [6] showing that it is much more convenient.

The modal accepts `ShapeManager`, `UndoRedoManager`, the id of the HTML element for the modal, a list of manager identifiers with which the added elements should be registered, and a special object that describes the fields the modal should include. The description of how to define the fields can be seen in Section 4.5.9.

The dependency on the managers should be attempted to be eliminated in the future because it should not be the responsibility of the modal to add the shapes, only collect the data. The reason the dependency is included though is that adding the shape to the `UndoRedoManager` should not be done when the `shapeAdded` event occurs. That is because the `UndoRedoManager` should not include commands for elements added during the initial loading of a shape/furniture the user wants to edit.

How to implement a new modal can be found in Section 4.5.8.

Additionally, there is a modal that is opened before saving the given element to collect some additional data. This modal is used to get the shape the user wants as a top-down view of a saved furniture. For that, there is a class responsible for showing all the options for the top-down view. This class populates the list based on the list of shapes discussed in subsection 4.3.1. The thing to note is that it currently skips the polygon and arc shape. The reason for this is that in the database we use the id of the shape template. This means that if the default identifier for the polygon or arc is used, it always uses the default definition of the shape as is in the database. So, a polygon would always place triangles, and

[6] https://app.diagrams.net/

an arc would always place a half-circle. In the future, it should be addressed by allowing the user to define the shape directly in the modal.

Although the modals currently contain fields where the color can be chosen for the newly added shape, they currently serve as an indicator of which shape is newly added. It does not persist in the database. This is currently like this, so it is not confusing for the user if, for example, an item location is highlighted in red but the user defined the element with the same color that is used for highlighting. However, allowing the user to use his colors would be better.

4.3.3 Save Buttons for Editors

The classes for the save buttons are responsible for adding the event handler to the save button that sends the request to the backend to save the element.

Because the constructor for the furniture save button accepts many parameters. A builder class is used so that initializing the save button class is less error prone.

4.3.4 Undo/Redo for Scenes

The `UndoRedoInterface` class is responsible for initializing the Undo/Redo buttons, so it is possible to also use Undo/Redo using a mouse. However, these buttons are currently only present in the `Edit` drop-down menu. The proper toolbox section should be implemented in the future to better reflect other editors and thus bring the application closer to what the user is familiar with.

4.3.5 List Pages

This section introduces the pages for showing the lists of shapes, furniture, floors, and items.

Shape, Furniture, Floor All of these pages are implemented almost identically. They are all simple unordered lists. To not require the user to scroll infinitely, the list elements are rendered paginated, with the user having the ability to choose how many elements per page should be shown.

In the future, some optimizations may be required here. Currently, all items are always loaded into memory, which can cause memory problems if the user has a lot of items. Furthermore, the responsiveness may struggle with a lot of items, because currently every render has to prepare data from the whole dataset before showing only the specific items. So, every render, the whole list is filtered based on the specified order and only after that the specific items are extracted and used to render the list.

Item The item list is a bit more advanced. It is a table compared to the previous lists. It also allows for sorting based on some columns. The action buttons are compressed into a single column that opens a dropdown menu as shown in Figure 4.2. There was a problem with the way the dropdown is rendered when there is a small amount of items in the list. More about the problem and its solution is given in Section 4.6.9.

The modal to add a new item also implements a simple category creator, as shown in Figure 4.3. The creator currently reflects the simplicity of the categories in the database. There is no separate persistence of the categories other than the string columns in the item rows. The category list is populated by iterating all the loaded items and collecting the categories from there. That also means that if we create a new category, then a new item is created, and immediately deleted if the page is reloaded, the newly created category will not be present.

On removal of an item, all other item's that are `contained` within the deleted item are deleted as well. There is currently no indication about which item's will be removed. The reason why it deletes all associated items is that it more closely reflects what the user would expect. The other option of deleting only the selected item could be considered as an additional option to add in the future. This option can be currently simulated by going to the `View` where the item is located and manually removing them all from the zone. This makes it currently easier to move than doing the alternative, which is to delete only the selected item. That is because if the user wanted to delete all the items that were associated with the deleted item, he would have to do it manually. The problem is that currently there is no indicator in the item list to be able to tell which items were contained within, and so the user would have to have the knowledge himself, making it very tedious.

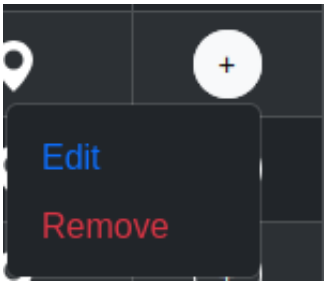


Figure 4.2 Action Dropdown

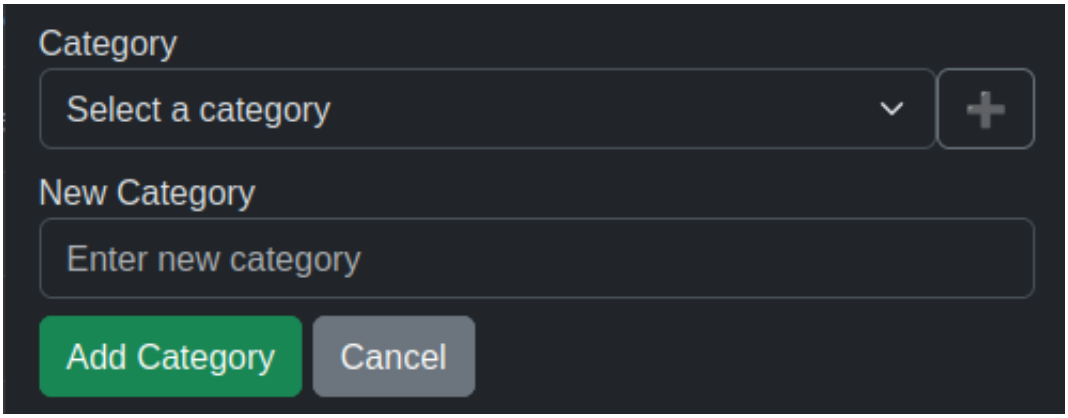


Figure 4.3 Category Creator

4.4 Backend

This section describes the implementation of the backend services.

The implementation of the management services follows a very simple logic. There is always a class acting as the controller of the application. It serves as an entry point into the service by implementing representational state transfer (REST) endpoints by annotating methods of the class with mapping, such as GET. After that, a Spring feature is used to implement the interaction with the database called repository. The classes with which the repositories are called entities. Entities should reflect schema of the database table as Spring can actually create the table in the database by itself, although in a production application Spring is set to validate that the schema as defined by the entity is identical to the schema in the database.

A new management service can be created as shown in Section 4.5.13.

The services require refactoring, so they are all up to the same standard. Because being a beginner in Spring and having no prior experience in Java, even though having experience in C#, there are many differences in the features used. As an example, usage of classes instead of records for simple data carrying classes, redefining types in the `FurnitureManagement` service instead of blindly treating it as JSON, since the service does not require to know the types.

All services also lack proper error handling, except the `ShapeManagement` service, which outlines a solution. This should definitely be solved in the future, as the user does currently have no way to understand what went wrong, although it is shown that something went wrong.

The API of the services is designed so that always only a single request comes from the user, and only on the backend will the data be redistributed accordingly. The same is attempted to be done also for inter-communication of the services, where batch operations were created, so it is possible to send a single request. However, it is not always possible to send a single request, as there are times when the returned data has to be properly paired, and in that case getting a list of the saved data, such as the ids, is insufficient.

Currently, if some update operation happens on some of the service. There is no indicator for the service if some of the elements, such as a zone in the furniture, was really modified. Currently, it leads to some wasted memory as the previous record could be reused.

Some of the API endpoints, such as the batch GET endpoint for items, are allowed to return deleted items. That is, so that the other dependent tables can be correctly cleaned up. These endpoints are publicly exposed, which is not really desired. They should probably be renamed and the access to the endpoint should be limited to a given service.

When the user deletes a specific element, all versions of that element are also marked as deleted. That is because the old versions can still be referenced by other elements, so the element does not have broken textures. It is then required to mark all the elements as deleted so it is possible in the future to implement some kind of migration system that would allow the user to get rid of the old versions in the element, etc. However, this leads to the problem that it is impossible to do a proper rollback of the database based on the versions only because it is impossible to tell if the version that is currently marked as deleted was actually still active. It should not prevent the user from working with multiple versions of an existing element if such a feature were requested in the future.

Shapes

This subsection explains some details about the `ShapeManagement` service.

The `ShapeInstance` has columns for the id and version of a shape. The joining of the shape is based only on the id. The version is intended for the possibility to warn the user that the instance of the shape he is using is using an old version of the shape.

The API endpoints for the shapes accept a list of instructions as a parameter. This helps validate that the instruction that is passed from the frontend is valid. The problem is that the instruction in the database is saved in JSON format. The solution on how to solve this problem is in Section 4.6.11.

Furniture

This subsection explains some details about the `FurnitureManagement` service.

Currently, to not require the `ItemManagement` service to send requests to the `FurnitureManagement` service, the item's are removed from the `ZoneInstances` lazily when the corresponding `FurnitureInstance` is accessed. The problem is that the item's are removed from the `ZoneInstances` inplace. That means if the item gets restored in the future, the information is lost that the item was ever in this version of the `FurnitureInstance`. Versioning of the `ZoneInstances`, or other alternative, should be considered so the information is correctly persisted.

Because there can only ever be a single specific `FurnitureInstance` placed on the floor. If the user moves the instance within the floor, the only thing that needs to be changed for the instance is the top-down view. For that reason, the history of the instances is kept in a separate table. It is also unlikely that the user would ever need to roll back the placement of the instance within the floor because if needed the instance can be easily moved back to the original location using the editor.

Floor

This subsection explains some details about the `FloorManagement` service.

The corners and walls of a floor are recreated for every update of the associated floor. That is mostly unnecessary, as the corners and walls are the elements that change the least in any given floor. For that reason, only updating the corners and walls that actually changed would be preferred.

Items

This subsection explains some details about the `ItemManagement` service.

To not require to rebuild the whole item tree to correctly reflect the item version, the item's parent is not saved with a specific version. It requires manually joining the item's using Spring rather than Spring being able to do it itself.

The sequence diagram shown in the Figure 4.4 displays the interactions of the `ItemManagement` service when updating an item.

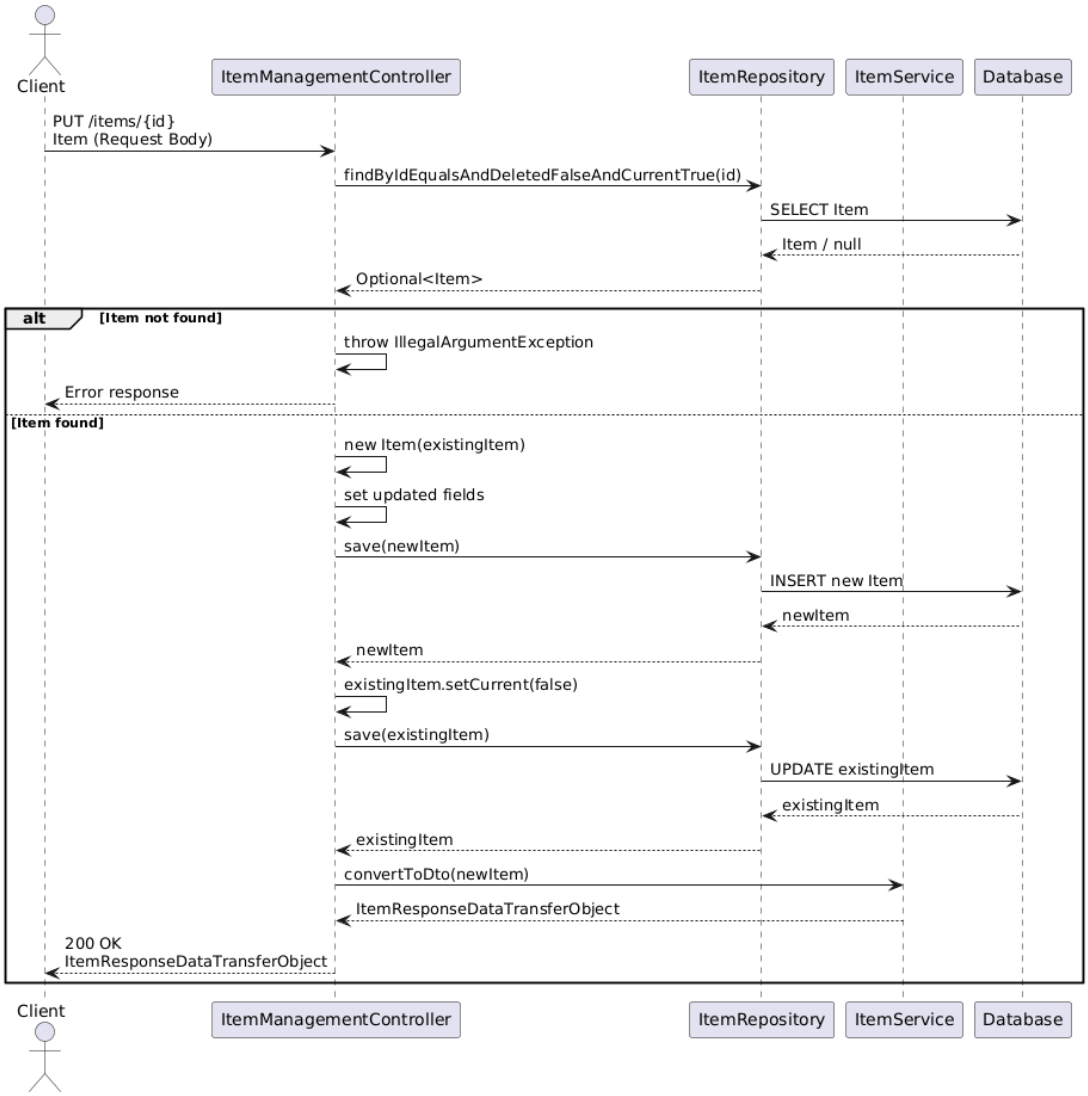


Figure 4.4 Update Item Sequence Diagram

Service Discovery

Implementing the `ServiceDiscovery` is really simple using Spring framework as it requires writing no code at all. It has to be configured using only the `application.properties` file as can be seen in Listing 4.8.

Other services can then easily register themselves by including `eureka.client.serviceUrl.defaultZone=http://${EUREKA_URL}/eureka/` in their own `application.properties`.

Listing 4.8 Service Discovery Configuration

```
spring.application.name=servicediscovery
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
logging.level.com.netflix.eureka=OFF
logging.level.com.netflix.discovery=OFF
```

Gateway

The gateway is also simple to implement using Spring. It just requires configuring a builder with paths, and it takes care of everything else by itself. Example of how to create a path can be seen in Section 4.6.12.

4.5 API

This section focuses on describing the design of the different interfaces of the application and how to extend the application.

4.5.1 Undo/Redo

To avoid direct dependencies on the manager, other managers do not interact with the manager directly. Instead, `CreateCommandEventHandler` classes are used. These handlers intercept events posted to the scene by various managers and translate them into commands that are then pushed to the Undo/Redo manager. This makes it so that new managers are not required to depend on the `UndoRedoManager` but instead a new `CreateCommandEventHandler` and commands can be created independently to add undo/redo functionality to the new manager.

The `CreateCommandEventHandler` class serves as a superclass and applies the template pattern. It defines the general workflow for intercepting events and creating commands while allowing subclasses to override specific behavior. For instance, the base class registers observers for move events and creates a `MoveShapeCommand`, passing it to the `handleMoveCommand` method. Subclasses such as `LabeledCreateCommandEventHandler` override this method to wrap the command into a `CompositeCommand` before pushing it to the Undo/Redo manager. Generally, to add a new manager for undo/redo, the `CreateCommandEventHandler` should get a new private method that registers listeners for the events of the manager, and for the event finished action, it should call a public handle method that can then be overriden by children to create a composite command if needed. It is also possible to create a completely new event handler that works based on this logic rather than extending the `CreateCommandEventHandler`.

4.5.2 Add New Shape Types

The new shape types added should inherit from `Phaser.GameObjects.GameObject` and additionally implement the `createSnapshot` method mentioned in Section 4.2.5.

4.5.3 Add New Shape Manipulation Manager

Creating new manipulation managers should follow this convention. First, there should be a `create` method that is responsible for activating the wanted functionality for a specified shape. Then, there should be an `update` method that can forcefully update the specified shape. It can, for example, update the handles for the shapes or forcefully change the position of a moved shape as explained in

Section 4.2.9. There should also be a hide method implemented that can be used, for example, to hide the displayed handles. It should technically be the reverse of the `create` method. These managers actually inherit from an abstract parent class `Manager` but such classes stopped being implemented after as it does not help in anyway in creating new classes because of the usage of JavaScript.

4.5.4 Add New Camera Manager

Because JavaScript is used, there are no interfaces (API) to create new camera managers, and although it is possible to create a parent class from which the camera managers could inherit, JavaScript does not enforce anything. For that reason, a certain convention is followed that new managers should implement. Every manager should start with a `create` method. This method is used to initialize all the required event handlers, etc. The initialization could technically occur in the constructor, but using the `create` method more closely follows the convention of other managers, such as managers, for manipulating the shapes where the create method is required, as can be seen in the constructor. shape manipulation section 4.5.3. After that, if required, the manager can implement an `update` method in case it has a functionality that is required to be called on every frame without being able to create a handler for it. Lastly, technically, every manager should implement a `destroy` method that cleans up all the used resources of the manager. Currently, the `destroy` method must be implemented only for managers who work with HTML elements, and also for managers who work with HTML elements, and if there are multiple scenes between which switching occurs and HTML is reused. The reason why it may not be required can be seen in Section 4.6.6 and in Section 4.2.13 it can be found that the `ScrollbarManager` is required to have the method implemented.

4.5.5 Add New Shape Creation Manager

To create a new shape creation manager, the manager should generally implement a method to add a new element to the manager, a method to create the element from a snapshot of the element for undo/redo functionality, and a method to remove the element from the manager. Additionally, if needed, it can implement a method to get a specific element or check if the manager already contains a specified element. There should also be another manager implemented that serves as a mediator between the creation manager and the undo/redo manager, so that the creation manager does not depend on the undo/redo manager.

4.5.6 Register New Shape

The `ShapeManager` uses the Registry pattern to register all the shapes that the manager should allow to create. For example, the `ShapeEditor` scene currently has registered a rectangle, arc, polygon, ellipse, container, and custom.

To register the shape, we use the `registerShape` method. First, it accepts a unique identifier for the shape type. Second, it accepts a factory method, which specifies how to create the instance of the shape. For example, the custom shape type is responsible for fetching the shape from the database and then uses the

manager again to reconstruct an instance based on the fetched instructions. Lastly, it accepts a configuration object called metadata that supports these options. It can be used to specify the default interactive configuration of the shape. This is used to enable interaction for the shape, and it is also used to specify the hit area of the shape. Then, a command string can be specified. This specifies how the instruction command for the shape will be called when serializing/deserializing the shape into instructions. For deserializing, the priority option might have to be specified, so if two registered shapes serialize into the same command, it chooses only a single shape into which it should serialize. It is used, for example, in the `ShapeEditor` to deserialize a `CREATE_CONTAINER` command into the container type, rather than the custom shape type that also serializes into a container. Lastly, a field map object can be specified. It is used to map names of fields of the shapes into another name. It helps to modify the given shapes so that they can be correctly saved in the database. As an example, the polygon has the point field in Phaser, but our database expects the polygonPoints field, so the field map defines this mapping.

4.5.7 Adding a New Scene

New scenes should always follow the scene creation system as specified in the Phaser documentation[7] and should implement some of the methods specified here[8]. Usually, the `create` method should always be implemented.

4.5.8 Adding New Element Modals

The modal class implements three methods. The `openShapeModal` method is responsible for opening the modal and rendering the fields. The method `onModalConfirmation` specifies what should happen when the user presses the confirm button. By default, it checks the validity of the fields and prepares the parameters to pass on. That is because for example, the user is expected to add the shapes with the origin at 0,0 because it is simpler for the user but internally the center of the shape is used as the origin. Lastly, there is a `handleAddShape` method that based on the template pattern is intended to be overridden to change the functionality of the `onModalConfirmation` method. It gets the shape type and all the parameters from the modal and adds the shape to the editor, additionally pushing the returned command to the `UndoRedoManager`. This is, for example, used in the `LabeledShapeModalUserInterface` that additionally adds a label to the shape and creates the composite command so that the label and shape are removed simultaneously.

4.5.9 Defining Modal Field Schema

The definition of fields is a key-value pair, where the key is the type of shape, and the value is an array of schemas for the input fields. The Listing 4.9 shows an example of the definition used to add an unlabeled shape to the editor.

[7] https://docs.phaser.io/phaser/concepts/scenes#scene-creation
[8] https://docs.phaser.io/phaser/concepts/scenes#scene-callbacks

Listing 4.9 Field Schema Definition Example

```
const ShapeFieldSchemas = {
  RECTANGLE: [PositionSchema, RotationSchema,
    DimensionsSchema, ColorSchema],
  ELLIPSE: [PositionSchema, RotationSchema,
    DimensionsSchema, ColorSchema],
  ARC: [PositionSchema, RotationSchema,
    RadiusSchema, ArcAngleSchema, ColorSchema],
  POLYGON: [PositionSchema, RotationSchema,
    PolygonPointsSchema, ColorSchema],
  CUSTOM: [PositionSchema, RotationSchema,
    DimensionsSchema, ColorSchema],
};
```

The Listing 4.10 shows an example of an input field schema. The type can be any of the types allowed for the HTML input. Additionally, the type can be a `group`, which groups multiple inputs together, or `points`, which create multiple inputs where the user can input multiple points. It is used, for example, to define the points of polygons. Figure 4.5 shows the modal for adding polygons. It also shows the group for the position. This definition also explains why it is not currently straightforward to add new shapes to the center of the screen. As they are rendered dynamically, it must be ensured that the fields are rendered in HTML before changing the initial value of the position.

Listing 4.10 Input Field Schema Example

```
const PositionSchema = {
  type: "group",
  name: "position",
  fields: [
    {
      type: "number",
      name: "shapeX",
      label: "X Position",
      attributes: { required: true, value: 0, name: "x" },
      validation: [{ event: "change",
        handler: validatePosition }],
    },
    {
      type: "number",
      name: "shapeY",
      label: "Y Position",
      attributes: { required: true, value: 0, name: "y" },
      validation: [{ event: "change",
        handler: validatePosition }],
    },
  ],
};
```

4.5.10 Adding New Backend Repository

Repositories are responsible for communicating with the database. They are implemented by creating an interface that inherits from a Spring repository interface. This interface is then used, and Spring injects the methods that work with the database. New interface methods can also be implemented where the

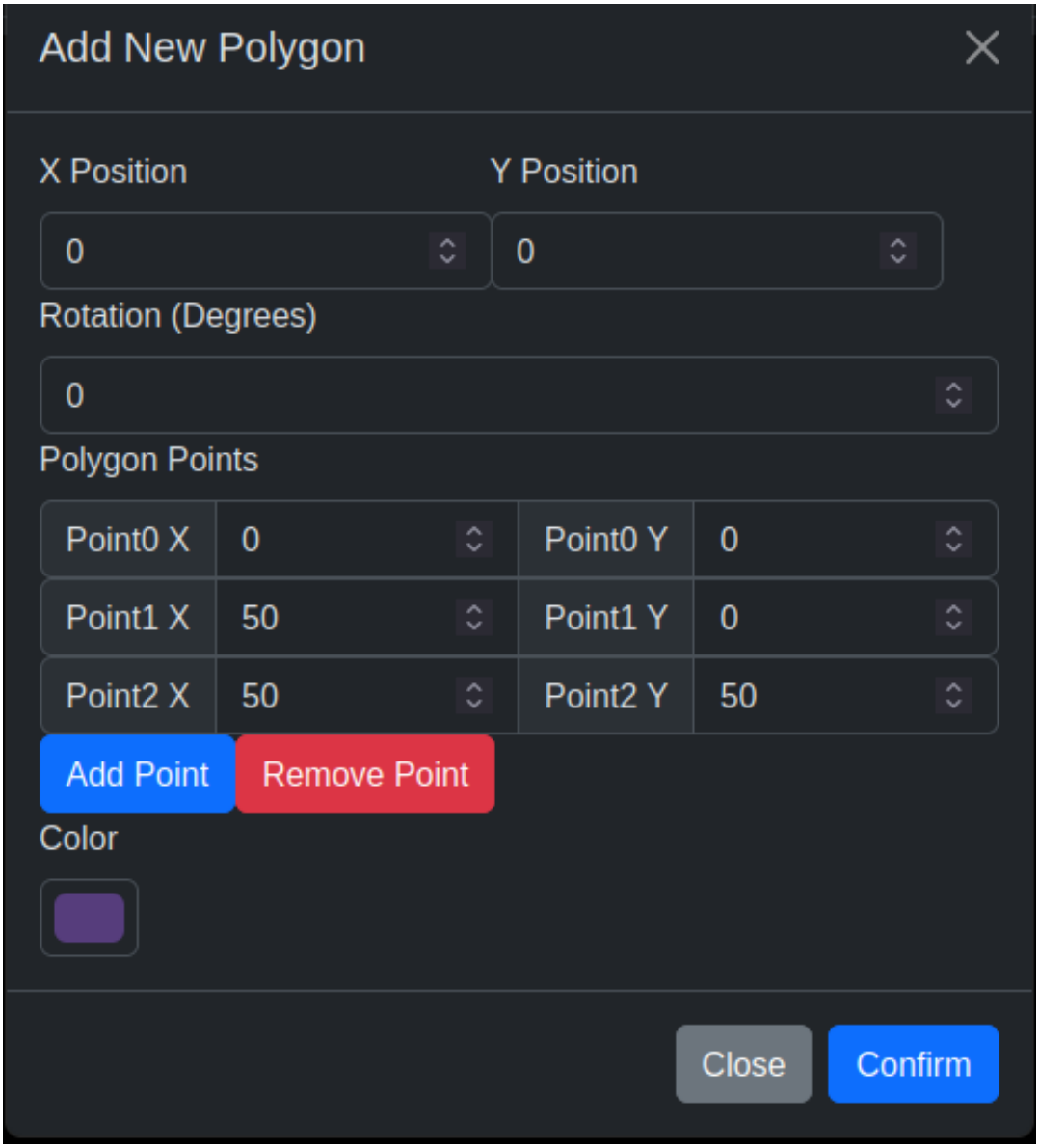


Figure 4.5 Modal for Adding Polygons

name of the method then implements a specific interaction with the database, as can be seen in Listing 4.11. If a more complicated interaction with the database is needed, the interface method can be annotated with `Query` and specify the desired SQL as can be seen in Listing 4.12.

Listing 4.11 Repository Method Example

```
boolean existsByIdEquals(Long id);
```

Listing 4.12 Repository SQL Query Example

```
@Query(
  """
  SELECT si FROM ShapeInstance si
  WHERE si.shapeId = :shapeId AND si.isTemplate = true
    AND si.shapeVersion = (
      SELECT MAX(subSi.shapeVersion) FROM ShapeInstance subSi
      WHERE subSi.shapeId = :shapeId AND subSi.isTemplate = true
```

```
    )
  """)
Optional<ShapeInstance> findLatestTemplateByShapeId(
  @Param("shapeId") Long shapeId);
```

4.5.11 Adding New REST Controllers

Controllers in Spring are classes that are annotated with `@RestController`. It then implements methods that are annotated with the specific REST method, for example, `@GetMapping("/api/path")`

4.5.12 Adding New Backend Entities

Entities in Spring are classes annotated with `@Entity`. They reflect the tables in the database. The columns of the tables are defined by creating properties of the class that are annotated with `@Column`. The key of the table is then annotated with `@Id`

4.5.13 Creating New Management Service

To create the new service, the new Spring application should contain atleast one controller which can be created as described in Section 4.5.11. Next, it should implement an entity that represents a table in the database which can be implemented as shown in Section 4.5.12. Lastly, it should implement a repository that is used by the controller to save the entity to the database and can be implemented as described in Section 4.5.10.

4.6 Implementation Problems

This section focuses on describing all the different problems that were encountered during the implementation and how they were solved.

4.6.1 Undo/Redo

At first, there was a problem that the `UndoRedoManager` was coupled with a specific scene to post its events. This became a problem when multiple scenes were introduced, between which switching is possible but the manager should work globally. For that, the manager was refactored to work as a publisher, to which others can subscribe.

4.6.2 Getting Correct Position And Size

There were difficulties getting the correct position and size of elements, which led to the implementation of the methods shown in Listing 4.5.

The approach used to calculate the dimensions reflects an earlier misunderstanding of transformation matrices. At the time of implementation, the behavior of Phaser's `displayWidth` and `displayHeight` properties was misunderstood, which led to manual calculation of width and height using the centers of the edges.

It is now understood that this behavior arises because the child's transformation is influenced by the parent's transformation without affecting the child's local transformation matrix.

Similar issues affect the calculation of the child's position. Although the current implementation works, it is computationally inefficient. Specifically, repeatedly calling `getWorldTransformMatrix` is expensive as it always recalculates the whole hierarchy by multiplying the parent matrices. A much more efficient approach would be to reuse the parent's world transformation matrix and use it to calculate the child's transformation matrix.

4.6.3 Fixed World

The development started with the assumption that the user would work with a fixed screen size as his work area inside the editors. That is the reason why adding new elements to the editor still works by using the modal window as described in the section 4.3.2. Using the modal at the time was not problematic, as the user could easily keep track of the coordinates. Now though, it would be better to place the element to the center of the camera and include a menu where the user could change the element's properties afterwards. The problem came when trying to implement the adding of shapes, so it is impossible for the user to add a new element outside the bounds of the screen. On every add to the editor, a check was made if the shape is outside the bounds, and in that case it was forced inside the bounds. This behavior worked but became problematic when the user tried to add shapes of sizes that were close to the size of the screen. Then the placement of the element became unintuitive and felt very awkward. That is why the decision was made to implement an infinite world instead, as explained in Section 4.2.13. Having an infinite world also solves a problem with inconsistencies when users use different resolutions of screens, as it would require doing some scaling of the added elements to fit the content of a user with higher resolution to a screen of lower resolution.

4.6.4 Using Native Browser Scrollbars For Moving Camera

HTML does not allow you to arbitrarily add browser scrollbars to any desired position. It enables to add them by defining an overflow property in the cascading style sheets (CSS). However, for that to work correctly, it requires encapsulating the HTML canvas element in a container element which would do this. cause the size overflow and the scrollbars to appear. The problem is that the canvas element does not change the size together with the container, and there were issues trying to figure out how to make that work correctly. To solve that issue the scrollbars are instead simulated by using the HTML input range element as explained in Section 4.2.13.

4.6.5 String Identifiers In The Shape Manager

The `ShapeManager` used to work on the basis of generated string keys. The commands always saved the identifier of the shape it should manipulate and queried the `ShapeManager` to get the shape it should manipulate. That is why

it was important that if the shape was deleted and afterwards created again, it would have the exactly same identifier for the commands to work correctly. There were a lot of problems with this approach, such as re-registering the shapes with their respective managers after deletion, among other things, so the manager was changed to work with references, which the commands save. For the references to work, though, the shape removal had to be changed to soft delete, where it just disables rendering and interactive capabilities on the shape. Restoring the shape in the commands is then a simple restore on the shape.

4.6.6 Scene Not Resetting Upon Restart

There is an issue in Phaser that even if a complete restart of a scene is performed, the state the scene had saved is not completely reset. This issue is mentioned in Phaser's documentation[9]. It is required to reset the scene state manually, otherwise unwanted behavior might happen, such as event handlers trying to operate on camera that no longer exists.

Currently, to circumvent this requirement, Phaser is forced to completely remove the scene from its internal manager and is then added back as shown in Listing 4.13. However, this should be addressed in the future and not be a permanent solution, as it forces an expensive reallocation of all the previous resources rather than reusing them. Furthermore, some managers are required to be correctly cleaned up anyway, such as the `ScrollbarManager`, mentioned in Section 4.2.13, where the event handlers registered for the scrollbars have to be removed, as the scrollbars are not managed by Phaser but are in raw HTML. In general, every event handler that should work for a specific scene but is on a HTML element should be removed. Another example is an event handler introduced for the HTML canvas element for view scenes explained in Section 4.2.2.

Listing 4.13 Code performing complete scene removal

```
this.events.on("wake", () => {
  this.scene.get("FurnitureView").events.once("destroy", () => {
    this.scene.add("FurnitureView", FurnitureView);
  });
  this.scene.remove(this.scene.get("FurnitureView"));
});
```

4.6.7 Dragover Event Not Triggered In Phaser When Dragged From Outside Phaser

Phaser has a `dragover` event that can be listened for on the editor objects. The problem with using the Phaser's event is that they do not get triggered when the canvas is not currently focused. To solve this, the `dragover` event is implemented on the canvas itself. The position of the pointer over the canvas is then calculated, and the Phaser's internal pointer can be forcibly changed to those coordinates. The Phaser pointer then allows for checking for overlaps with editor objects where the top-most is chosen as the target. An example implementation of a handler using this logic can be seen in Section 4.2.2.

[9] https://docs.phaser.io/phaser/concepts/scenes#problems-restarting-a-scene

4.6.8 Undo/Redo For Item Movement

The movement of items is implemented by dragging HTML elements as mentioned in Section 4.2.2. This brings about problems when trying to implement undo/redo functionality for the elements in the zone lists. If undo/redo occurs on top of being required to restore the data in the item map and in the zones to their previous state, it is required to properly restore the state of HTML. The problem is that it has to happen as a composite command, and the undo/redo should also work while the modal window for the zone is not open, or if the window for the other zone is opened. In that case, the HTML that the commands use does not exist and nothing should happen, but even when we close and open the same modal window, the HTML inside the modal is technically different since it is dynamically rendered. To solve that, a query selector is used that gets the specific element on request instead of saving a reference to the original object. An example of this is shown in the Listing 4.14. Another problem that also arises is that simply re-adding a cloned HTML element is not enough to ensure the previous functionality. If an element is cloned, it does not also clone the registered handlers of the element. That means they have to be registered again.

Listing 4.14 Querying HTML Element Using A Selector

```
const elementSelector = itemData.parentId
    ? `ul[data-parent-id="${itemData.parentId}"]`
    : `ul[data-zone-id="${itemData.zoneId}"]`;
const parentList = document.querySelector(elementSelector);

if (!parentList) {
    return;
}

const itemElement = parentList.querySelector(
    `li[data-id="${itemId}"]`,
);

parentList.removeChild(itemElement);
```

4.6.9 Item List Page Action Button Dropdown Render Problem

Since the dropdown menu is a child element of the table, it caused an issue with the way the dropdown was rendered when there were less than 3 items in the list. That is because the list is made scrollable and if there is a small amount of items, the dropdown menu's size is bigger than the height of the remaining rows. That caused the overflow for the scroll to trigger. Then, to access the dropdown menu, the user would be required to scroll to it.

The solution to the problem is to check if the amount of items is small, and in that case the overflow on the table changed so that all the content is visible.

4.6.10 Issues With Composite Keys In Spring

There were many issues. trying to work with composite keys in Spring. Spring does not allow to use an automatically incrementing sequence for the id when the

id is part of the composite.

The first solution to that was to always check what the current max id is in the table and increment that. However, this solution was not adopted because there were concerns that it would not work correctly if multiple replicas of the service were to query the database at the same time.

Instead, the id was changed from a numerical identifier to a universally unique identifier (UUID). This works as expected, but the problem is that the identifiers are exposed to the user in the list pages, which makes UUID unattractive, additionally, even though the chance is small, there are chances of duplicates, and lastly, the main problem is that it breaks the code that sorts the elements based on time of creation.

The final solution is to manually create a sequence in the database and query it for the next id. This should work with multiple replicas, even if the replicas are on different systems.

4.6.11 Saving Data As JSON In Spring

If an object is accepted on the REST controller, that should be saved in the database as JSON. It is required to implement a custom converter that converts the object to the JSON format as a string and back from the JSON string to the object, otherwise it will throw errors when attempting to save it to a database. Spring by default uses the Jackson[10] library to work with JSON. By default, this library does not support converting time, which is how the versions are implemented. A `JavaTimeModule` extension has to be included to make the conversion work.

4.6.12 Adding New Path To Gateway

To create a new path for the gateway, the route method should be used in the route builder. The first argument is a unique route identifier, while the second argument is a lambda expression that specifies the route definition, including its path predicate, filters, and target URI. An example of a path-based route definition is shown in the Listing 4.15. The route matches requests starting with /furniture-management, rewrites the path by removing this prefix, and forwards the request via load balancing to the furniture-management service, where the name corresponds to the Docker service identifier.

Listing 4.15 Example Gateway Route Definition

```
.route(
    "furniture-management",
    r -> r.path("/furniture-management/**")
        .filters(
            f -> f.rewritePath(
                "/furniture-management/(?<segment>.*)",
                "/${segment}"))
        .uri("lb://furniture-management"))
```

[10] https://github.com/FasterXML/jackson

4.7 Testing

This section describes the approach used to test the application. Currently, there are no automated tests implemented. The primary reason for this is the dependence on the HTML canvas element, which makes traditional automated testing impractical. The canvas does not expose its internal drawing state, as the canvas on its own is just a bitmap, making it so there is no straightforward way to programmatically verify whether graphical operations have produced the correct visual output.

While snapshot-based regression testing is possible, it is not well suited for test-driven development as the snapshots can only be created once the feature is already functioning correctly. There is no simple way to test whether operations on the canvas are happening correctly. Only way to really test is to create regression tests that take snapshots of the application and afterwards compare the snapshots if there is any difference which would indicate something became broken, but those are not useful for test-driven development as the functionality already has to be correct. It would only help verify whether our new features broke our already implemented functionality and would simplify refactoring.

For that reason, testing was performed manually throughout the development by manually verifying that everything works correctly. This manual approach was feasible because the code is designed so that the components are as loosely coupled as possible. That makes it so that changes in one area of the code are unlikely to unintentionally affect other parts of the code. As a result, manual testing could be performed relatively efficiently and reliably during development.

In addition, the application was tested with one user to confirm that the application works. Among the other issues already mentioned prior, the user found that it is very inconvenient if a furniture, or floor was created while the editor was zoomed out, because upon loading the object the zoom is reset to default, which makes the object too zoomed in. This leads to issues when the user wants to use the hover over object feature with items, as the placement zone can take up the whole editor screen, leading to automatically opening the modal for that zone. Also, it is impossible to zoom out if the user is already holding an item because the scroll wheel event does not activate when the canvas is not focused, which is the case here, as the item is dragged from outside the canvas.

5 User Documentation

This chapter provides instructions on how to use this application. It starts with describing the minimal requirements for the application to run, it continues with the instructions on how to deploy the application, and it lastly explains how to use the application itself.

5.1 Administrator Handbook

This section focuses on describing the requirements for deploying the application. Afterwards, it gives instructions on how to deploy the application. Finally, It gives some troubleshooting steps in case of having problems with deploying the application.

5.1.1 Hardware Requirements

The following is a description of the minimal hardware requirements required for the machine. This assumes the usage of the predefined deployment using Docker. An advanced administrator can manually deploy each service on its own machine, making the requirements different.

Although microservice architecture is used, currently docker compose is used to deploy the application. This requires that all services run on a single computer, which requires a computer with more available resources to handle all the services.

Central Processing Unit (CPU) 2 GHZ dual-core 64-bit CPU. It is important for the CPU to support virtualization and it has to be enabled in the basic input/output system (BIOS).

Memory Minimum 16 GB of random-access memory (RAM). However, more is recommended, as running multiple Docker containers is quite memory heavy.

Network Card As this is a web application, any network card is required to be able to run the application.

Disk Space 10 GB of storage should be enough for minimal deployment, but the required storage space depends on the amount of data saved by the users, so preparations for extending memory should be prepared if needed.

5.1.2 Software Requirements

The following is a description of the minimal software requirements required for the application to run. This assumes a predefined deployment using Docker. If you do not use Docker, the requirements are different.

Operating System The application supports all major operating systems, such as Windows, Linux, and macOS. Generally, it can be deployed on all system being able to run Docker.

Docker The Docker Engine, but also Docker Compose is utilized to deploy the application.

Web Browser Any web browser is necessary to access the user space of the application. The application should be browser-agnostic, but only Chrome and Firefox are officially supported.

5.1.3 Deployment

This section focuses on describing how to deploy the application. It presents the commands that deploy the application and also mentions some troubleshooting instructions.

As already mentioned in Section 5.1.2, Docker with Compose is required to deploy the application.

First Step – Move to the project directory First, it has to be ensured that the currently selected directory is the top-most directory of the project that contains the main docker compose files. It can be done by running the following command.

```
cd Warehouse-Manager
```

Second Step – Changing API URL It is important to change the `API_URL` variable in the `services/gui/src/config.js` file. The address should be changed to a real address by which the machine on which the application is deployed can be identified. For testing, the API is exposed only on the localhost which means if a user connects to the application from a different machine, he will not be able to save any data.

Third Step – Building Docker Images All required Docker images should be built. This step usually takes a long time on the first run as it requires downloading the base images for all the other images. Additionally, it runs all commands defined in the service Dockerfiles. Running this command afterwards is usually much faster, as the base images are downloaded and Docker uses a cache to prevent rerunning commands where no file was changed. The following commands builds the Docker images.

```
docker compose build
```

Fourth Step – Running the Application Starts all the services defined in the Docker Compose file. On first execution of the command, it has to first download all the dependencies in the associated containers, which leads to longer starting times. The following command is the command to run the application.

```
docker compose up
```

After the command finishes, these containers should be visible in `docker compose ps` output: item-management, furniture-management, floor-management, shape-management, shape-database, floor-database, item-database, furniture-database, gateway, service-discovery, user-interface. The application interface should then be able to be accessed by accessing the `http://localhost:8084` URL in a browser, though it is better to test from a different machine by using the machine's internet protocol (IP) address with the specified port. The main page shown in Figure 5.1 should be seen. It is also recommended to go through the buttons on the home page and for every loaded page check if there is no error in the browser console which can be accessed by pressing F12.

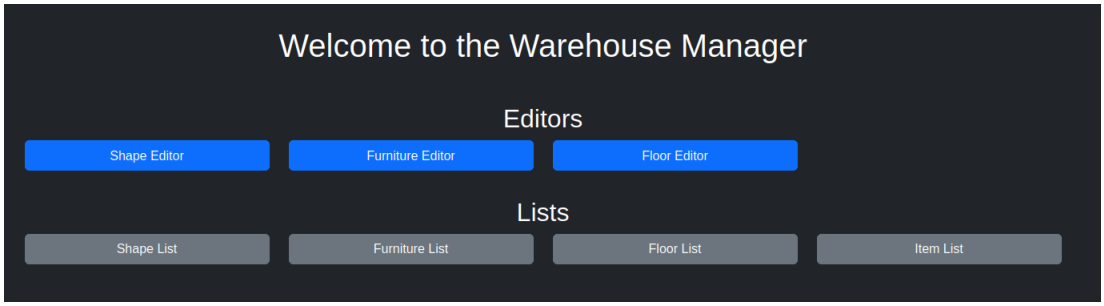


Figure 5.1 Home Page

Troubleshooting

- Is the Docker daemon running? – Ensure that Docker is correctly installed and running.
- docker compose not found – Ensure that V2 of Docker compose is correctly installed.
- Address already in use/port already allocated – Means a port conflict on the machine. Change the conflicting service port by changing the port values in the `docker-compose.override.yaml` file.
- Errors during build process – If image build process fails the issues can be identified in the terminal before the command fails.
- Errors during start process – Check logs of the container by running the following command

  ```
  docker compose logs some-service
  ```

  The following text should be visible in the logs for databases:

  ```
  database system is ready to accept connections
  ```

  The following text should be visible in the logs for management services, as it usually means the service is working and is correctly registered with the service discovery.

```
com.netflix.discovery.DiscoveryClient : \
  The response status is 200
```

The following text should be visible in the logs for the service discovery:

```
o.s.web.servlet.DispatcherServlet : \
  Completed initialization in 1 ms
```

- resources exhausted, etc. – Not enough hardware resources available to run the application. Make sure the machine follows the minimal requirements.

5.2 User Handbook

This section focuses on describing how to use the application. It describes all the pages that can be navigated to, their screen, and some keyboard shortcuts if the page supports them.

To access the application, a browser is required. Virtually any browser should be viable to use as the application is browser-agnostic. However, there might be some irregularities across the browsers.

5.2.1 Home Page

Figure 5.2 shows the main page of the application. It serves as the main navigation point of the application, allowing access to all the separate parts of the application. The screen is separated into two sections: editors and lists. Each button then takes the user to the corresponding page.

- Shape Editor – Opens the Shape Editor where new base shapes can be created. More detail in Section 5.2.2.
- Furniture Editor – Opens the Furniture Editor where new furniture definitions can be made. More detail in Section 5.2.3.
- Floor Editor – Opens the Floor Editor where new instances of floor, or rooms can be created. More detail in Section 5.2.4.
- Shape List – Opens a page where all the user-created base shapes are listed and allows their management. More detail in Section 5.2.6.
- Furniture List – Opens a page where all the created furniture definitions are listed and allows their management. More detail in Section 5.2.7.
- Floor List – Opens a page where all the created floors are listed and allows their management. More detail in Section 5.2.8.
- Item List – Opens a page where all the items are listed and allows their management. More detail in Section 5.2.5.

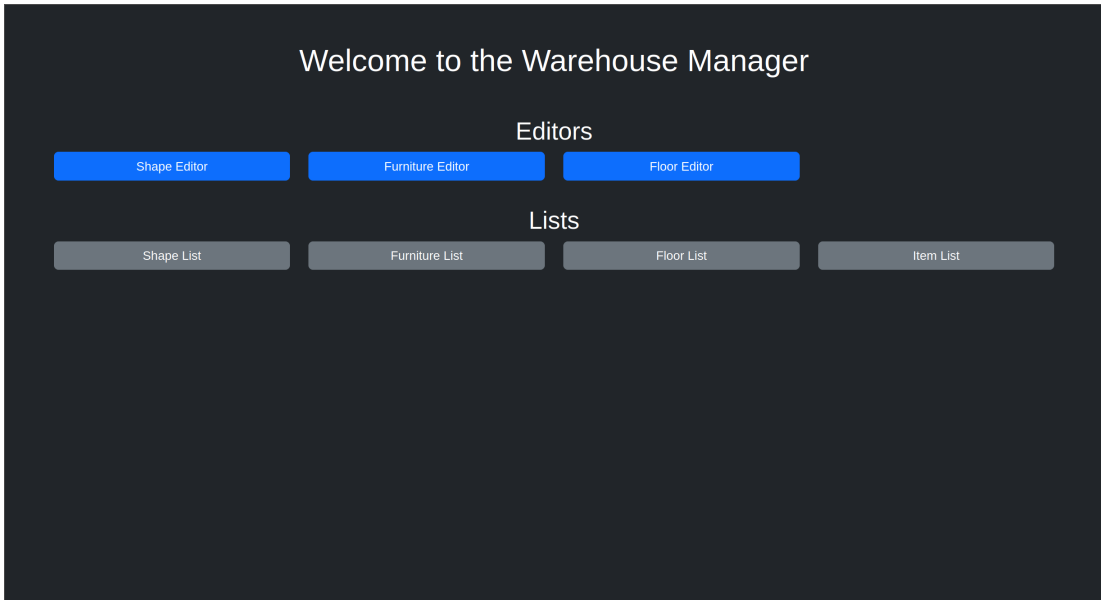


Figure 5.2 Home Page Screen

5.2.2 Shape Editor

Figure 5.3 shows the Shape Editor page of the application. On this page, new base shapes can be created in case the few existing system shapes are not sufficient to model a specific furniture.

The page is separated into 3 sections. The header (1), the left menu (2), and the main editor window (3).

The header contains 3 components: The edit dropdown menu (4), which contains options to undo/redo a previous action, a text box (5) to enter the name of the shape, and a save (6) button to save the shape.

The left menu is a list of all the base shapes that can be used to create other base shapes. Upon clicking on the base shape, a modal window is displayed where the base parameters of the shape should be filled. After filling in the parameters and pressing `Confirm` in the modal, the new base shape is added to the editor for further manipulation if needed.

The editor area allows for manipulation of the added shapes from the left panel.

Editor Controls Shape can be selected by pressing it. Selected shape has rectangular handles that can be dragged to resize the shape. Additionally, a circular handle is displayed that can be dragged to rotate the shape. Example of a selected shape can be seen in Figure 5.4. The shape itself can be dragged to change its position. The editor also has a vertical and horizontal scrollbars that can be used to pan the camera. Holding left click on an edge of the editor also pans the camera, allowing the camera to pan while moving or resizing a shape. On top of that, the Table 5.1 shows shortcuts that can be used to manipulate the editor, such as deleting a selected shape.

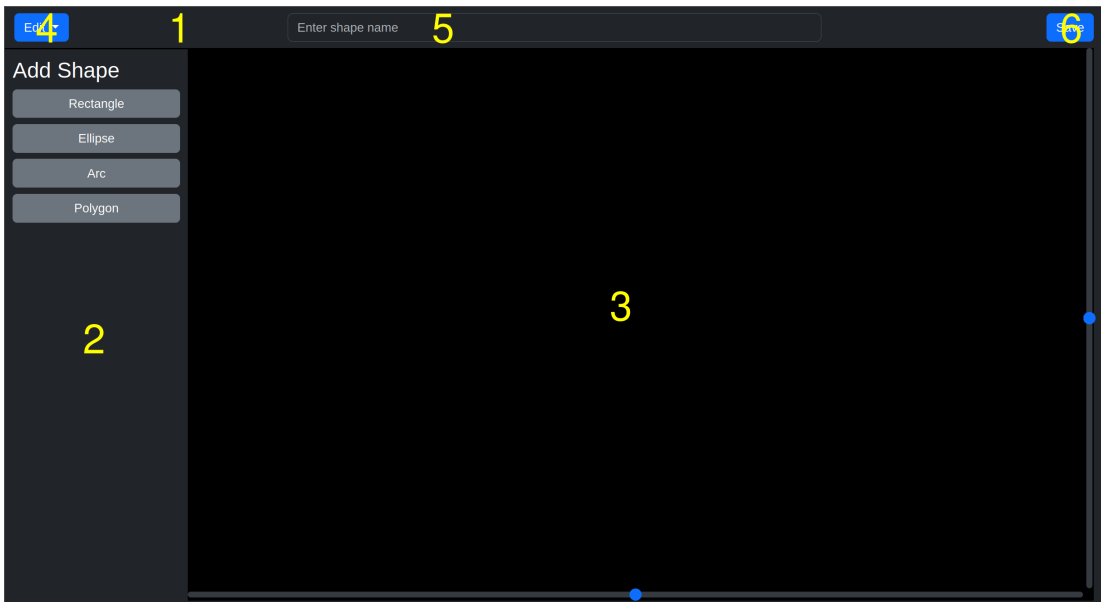


Figure 5.3 Shape Editor Screen

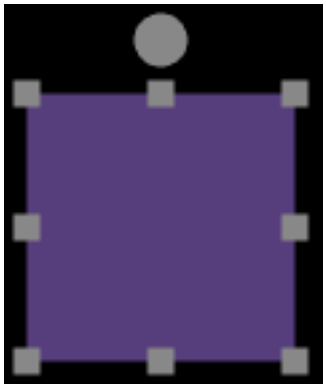


Figure 5.4 Selected Shape Example

5.2.3 Furniture Editor

Figure 5.2.3 shows the Furniture Editor page. On this page, a new furniture definition is modeled based on an existing real one in which items can be placed when the furniture is placed on a floor. For general editor usage, refer to Sec-

Device	Shortcut	Action
Keyboard	Ctrl + Z	Undo last action
Keyboard	Ctrl + Y	Redo last action
Keyboard	Delete	Remove selected element
Mouse	Scroll Up	Zoom in
Mouse	Scroll Down	Zoom out
Mouse	Middle Click (Mouse 3)	Pan view

Table 5.1 Keyboard and Mouse Shortcuts

tion 5.2.2, where it is already explained. This section will focus on differences from the Shape Editor.

The difference from the Shape Editor is that the header (1) of the page contains a switch (2) called `Add Zone`. When this switch is ON, adding new shape changes to adding a new zone where in default parameters it will additionally ask to select a name for the placed zone. Zones are places where items can be placed within the furniture. An example of a placed zone is shown in Figure 5.5. The zone within the editor is just a shape with a label on top of it. The name of the zone can be changed by double clicking on the label. Figure 5.7 shows an example of a simple shelf created in the editor.

Upon clicking the `Save` button (3), a modal window is displayed with a dropdown list where a shape representing the furniture while looking at the furniture from top should be chosen. This shape is how the furniture will be displayed when placed within a floor. Note that to use a polygon or an arc, the specific shape should be first modeled in the Shape Editor, as using an arc or polygon for the top down view is too generic.

Figure 5.5 Zone Example

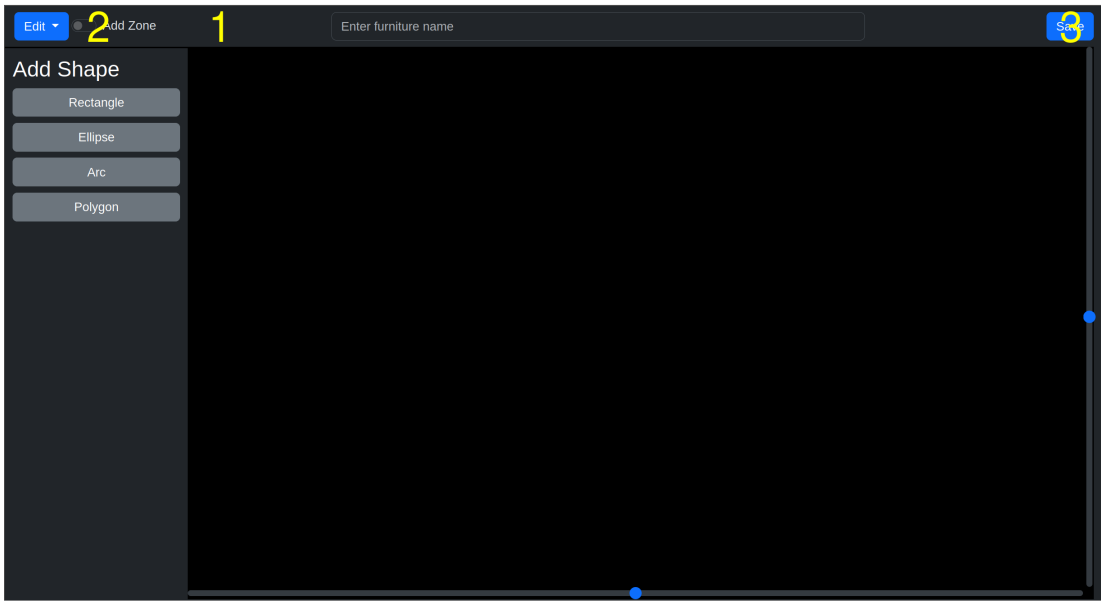


Figure 5.6 Furniture Editor Screen

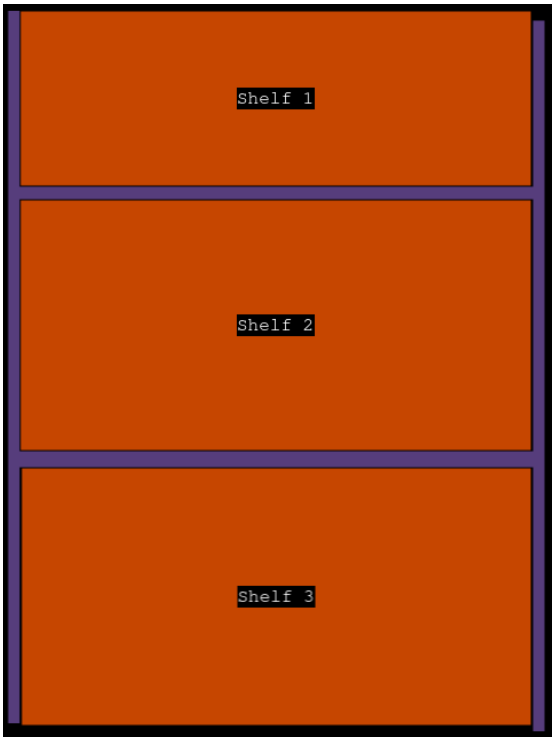


Figure 5.7 Furniture Definition Example

5.2.4 Floor Editor

Figure 5.8 shows the Floor Editor page. On this page, a specific room or floor can be modeled by first defining the corners, and afterwards adding a furniture definition, previously created using the Furniture Editor, can be placed to replicate the real life counterpart of the floor. For general editor usage, refer to Section 5.2.2. This section will focus on the differences from the Shape Editor.

Figure 5.8 Floor Editor Screen

The left menu (1), compared to the Shape Editor, makes it possible to add

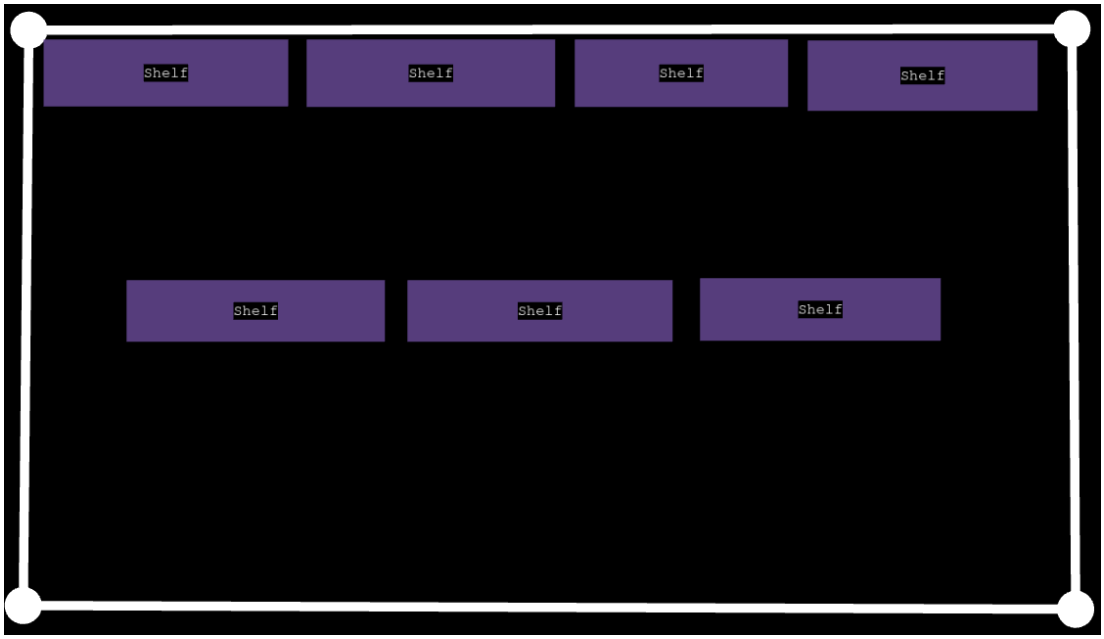


Figure 5.9 Floor Example

corners or predefined furniture. The `Corner` button (2) works as a toggle. When the button is turned on, when the mouse is dragged over the editor window (3), it shows a preview where the corner will be placed. Pressing left click places the corner. Dragging the corner to a new position still works, but, compared to the other editors, it is not allowed to resize or rotate. Turning the button of disables the placing of the corners. When two corners are selected, a wall is placed between those two corners. Dragging a corner with a wall attached also changes the position of the wall. Currently, a wall cannot be deleted directly, so an attached corner has to be deleted to remove the wall. Placing the furniture works the same as in the other editors, with the exception that there is a label with the furniture name on the shape to identify the furniture. Figure 5.9 shows an example of a floor created using the editor.

5.2.5 Item List

Figure 5.10 shows the Item List page. It lists all existing items and is used to create new items, update their information, or delete them.

It is displayed as a table. In addition to the columns describing the item itself, there is a column for location (1) that can be pressed and takes the user to the Item Location page, which is explained in more detail in Section 5.2.10, and the `Action` column (2), which contains a button that can be pressed to show a dropdown menu with buttons to remove or edit the item.

The `Add New Item` button (3) opens a modal window where information about the item can be filled in and added afterwards. To the left of the button there is a dropdown menu (4) where certain criteria on how to sort the items can be chosen. To the left of the dropdown menu, there is a dropdown menu (5) where you can choose the number of items per page. Lastly, left to the `Items per page` menu, there is a search bar (6) that can be used to search for a specific existing item.

Beneath these menus is the table section itself (7). The columns header (8)

can be pressed to force alphabetical sorting based on that specific column.

Below the table is a pagination (9) that allows for switching between the pages.

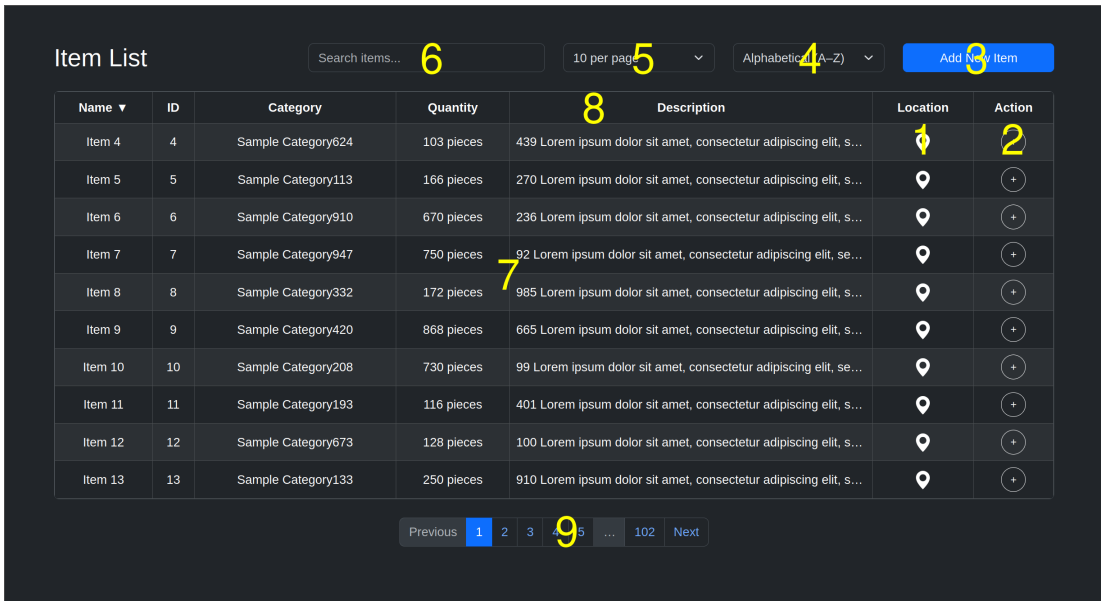


Figure 5.10 Item List Page

5.2.6 Shape List

Figure 5.11 shows the Shape List page. It displays the list of all the shapes created by the user and allows the user to remove the shape or allow the user to press edit (1), which takes the user to the Shape Editor 5.2.2 where the existing shape can be edited. The `Add New Shape` button (2) redirects the user to the Shape Editor 5.2.2 where a new base shape can be created. For other functionality refer to the Section 5.2.5 as this list is a simplified version of the Item List.

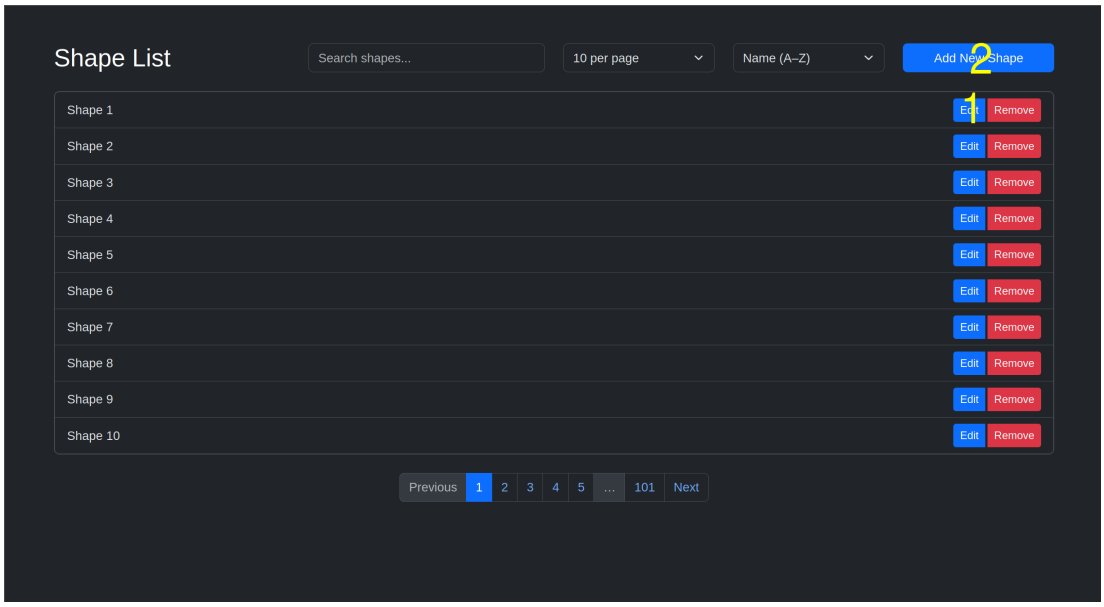
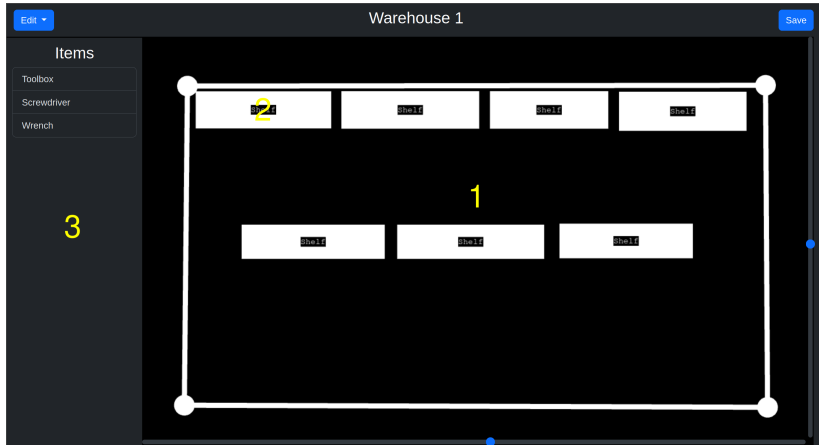
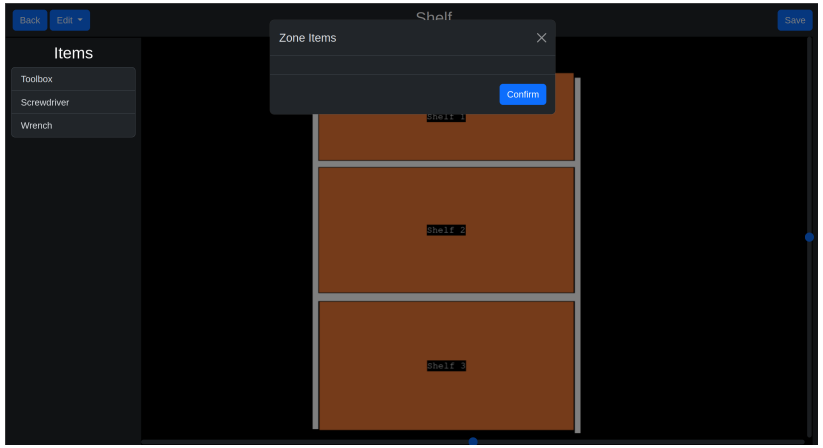


Figure 5.11 Shape List Page

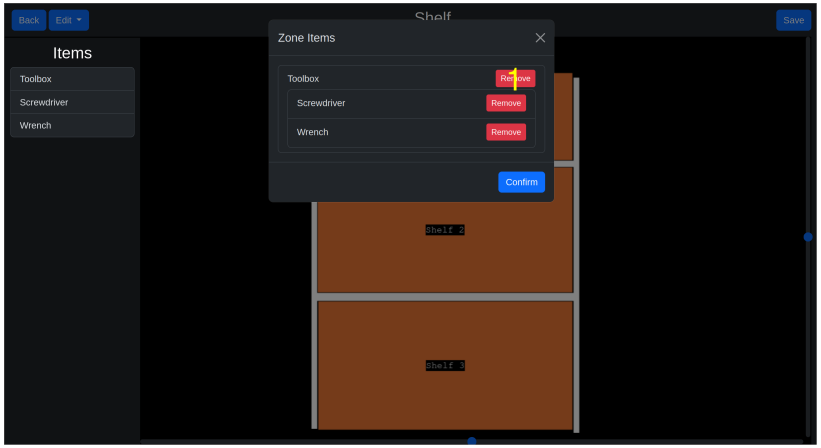


(a) Floor View Screen

(b) Furniture View Screen



(c) Zone Items Screen



(d) Zone with Items Example

Figure 5.12 View Screens

5.2.7 Furniture List

For the Furniture List page, refer to Section 5.2.6 as the Furniture List is an equivalent of the Shape List but it works with furniture instead of shapes.

5.2.8 Floor List

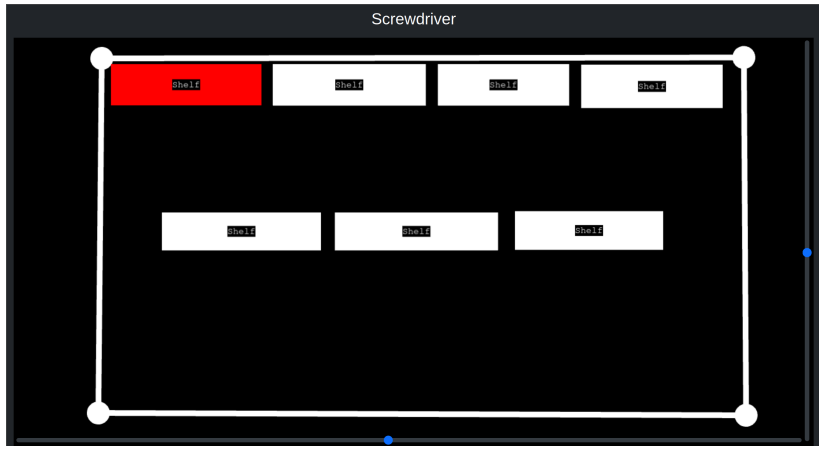
For the Floor List page, refer to Section 5.2.6 as the Floor List is an equivalent of the Shape List but it works with floors instead of shapes. In addition, every floor has a `View` button which can be pressed to take the user to the Floor View page where it is possible to add new items to the floor. More about the views is given in Section 5.2.9

5.2.9 Views

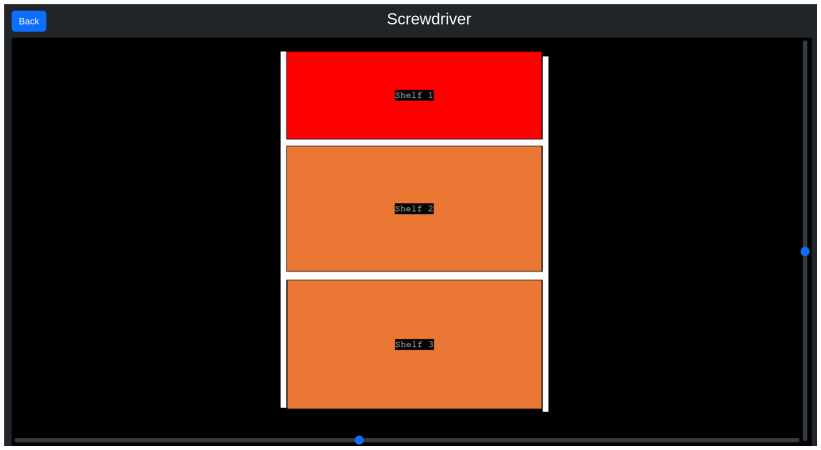
Views is a page that allows adding created items to specific locations within a floor. First, an unmodifiable version of a floor (1 in Figure 5.12a) is shown as can be seen in Figure 5.12a. The furniture (2 in Figure 5.12a) on the floor can be clicked, changing the view to a furniture view, as can be seen in Figure 5.12b. Lastly, a specific zone (1 in Figure 5.12b) can be pressed, which opens a modal with the items in the zone, as can be seen in Figure 5.12c. It is then possible to drag an item from the left menu (2 in Figure 5.12b) to add an item to the zone. Additionally, it is also possible to drag the items across other items to nest them. An example of a zone with added items can be seen in Figure 5.12d. To remove an item from a zone, the `Remove` button (1 in Figure 5.12d) can be pressed. To navigate back from the furniture view to the floor view, the `Back` button (3 in Figure 5.12b) in top left should be pressed. It is also possible to drag an item from the left menu (3 in Figure 5.12a)over a furniture and hover over the furniture for a while to open its view. Subsequently, the dragged item can also be hovered over a zone to open the modal for that specific zone and subsequently add the item to the zone. It is also possible to control the camera as described in Section 5.2.2 and use all the same shortcuts, including undo/redo, other than `DELETE`.

5.2.10 Item Location

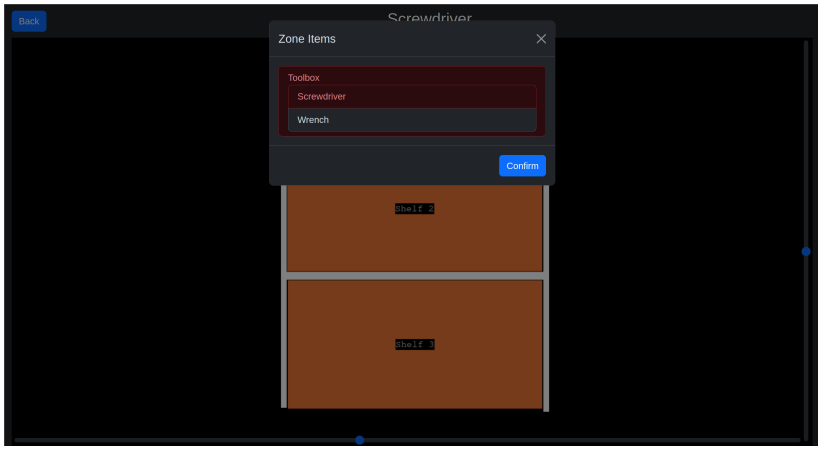
The Item Location page is a read only version of the Views where to highlight the location of the item, first the furniture, then the zone, and lastly the items are highlighted in red to show the final location. Figures 5.13a, 5.13b, and 5.13c show an example of a highlighted location within the view. It also applies that the camera can be manipulated in the same way as described in Section 5.2.2 and navigation through the views here is possible only by pressing the elements because hovering over the element does not make sense in this context.



(a) Item Floor Location Example



(b) Item Furniture Location Example



(c) Item Zone Location Example

Figure 5.13 Item Location Examples

Conclusion

A simple inventory management system has been implemented that enables users to organize and locate physical items within custom-defined environments. The system supports the creation of user-defined base shapes, furniture definitions, and floor plans that correspond to real-world layouts. Users can create items, place them within specific locations in furniture placed on floor plans, and move items between locations as needed. All major domains, including shapes, furniture, floors, and items, can be managed through the application.

Compared to using usual tools, e.g. Sortly or spreadsheets, where the only way to represent location is textual, the system allows to visually represent the location. This reduces errors because the user does not have to remember the textual description of the location, if the item is required to be moved between locations, it can be moved to the new location as if it was done in real life.

This brings advantages to small-scale companies because the system has a low barrier to entry. It does not require extensive training to use or any special knowledge about the company's internal sorting system, as the user can immediately visually see the layout of the warehouse and moving of specific items is implemented, so it as closely follows how the user would move the item if the item were in his hands.

The following is a list of goals that were introduced in the introduction. The list describes to which level the goal was achieved, how it is achieved, and some problems that had to be solved to achieve the goal.

- **Provide Real-Time Location Lookup** – The goal was fully achieved. Users can quickly search for a specific item in the item list and query for its location. After that, the floor where the item is located is shown with the specific furniture highlighted. The location can be traced down to the specific container within a piece of furniture. The biggest problem to achieve this goal was how to design the data so the item's location can be effectively found.

- **Support Custom Floor and Furniture Design** – The goal was fully achieved. Users can recreate real-world floor layouts and furniture structures using the implemented visual editors. There were many problems in trying to achieve this goal. This is because to implement an editor, there are quite a lot of functionalities that the users expect that are not trivially achievable but still have to be implemented. Examples are: zooming the camera, resizing elements, or infinite world.

- **Offer Reusable Libraries** – This goal was not achieved as the current implementation supports only a single-user environment, but the system is implemented in such a way that if multi-user environment is implemented, this goal should not be hard to achieve. Alternative functionality was implemented that can be found below this list.

- **Allow Item Management** – This goal was fully achieved. Items can be added, removed, edited, and moved between locations. Items can be added, removed, and edited on the item list page. Relocation of items is performed

through an intuitive visual editor that closely resembles how items are moved in real-world environments. The biggest problem with item management is the recursive nature of the items. It complicates the persistence of the item tree, the user must be allowed to place items within items without it being tedious, and the versioning mentioned below this list gets much more complicated, so it does not get completely broken.

- **Minimize Mental Load** – This goal was achieved. The system is designed to be usable without imposing a high cognitive burden. Its interaction patterns and visual structure aim to ensure that new users can quickly understand the functionality and that returning users can effectively use the system even after periods of inactivity. The biggest challenge was implementing editor functionality in a way that closely follows interaction patterns found in commonly used visual editors, so that users are not required to remember system-specific controls or workflows. Another challenge was designing the item movement feature in a way that is intuitive and closely resembles how items are moved in real-world environments, without requiring additional explanations or overly complex interactions.

The goal of offering reusable libraries was replaced by implementing versioning of selected domain entities, allowing users to modify existing definitions without breaking previously created data. This prevents issues such as broken references and enables recovery of accidentally deleted data. The reason for implementing versioning instead is that it was considered critical for the user experience. Without the library, the application is still usable just fine, but without the versioning, the user-experience would not be as good. As an example, if the user would have changed a furniture definition, if there was no versioning, every instance placed would get changed based on that definition. That is not how furniture really works. It would be very unusual to modify an already purchased furniture. With versioning, the old version is still used and if a change is desired, the user can place the new definition of the furniture and transfer all the items from the old instance to the new instance, which is also closer to how it would happen in real life with new furniture. Versioning will also allow one to implement migrations that the user would be allowed to trigger that would change the instances to the new versions, but there would be a much better control over the migration than the immediate change if versioning was not implemented.

Future development of the system could focus on introducing a multi-user environment with role-based access control, as originally planned. This would enable advanced features such as shared and reusable definitions, allowing inexperienced users to easily reuse work created by others. A multi-user setup would also make cooperative editing possible by synchronizing changes made by multiple users working on the same definition.

Another important improvement concerns the internal representation of items. Currently, item quantities and categories are stored only as strings, which limits filtering and organization. Introducing a structured category system and a more robust quantity model would allow a single item to exist in multiple locations with different quantities, better reflecting real warehouse use cases.

The system could also be extended with alternative methods for moving items,

such as using barcodes or QR codes instead of relying solely on visual manipulation in the editor. This would improve practicality in real-world deployments.

Several user interface improvements are also possible, including persisting element colors, reducing the use of modal dialogs, improving navigation, and adding clickable buttons for actions currently available only through keyboard or mouse gestures. In addition, the application could be expanded with inventory management features such as reports and alerts, which are currently missing due to the focus on location tracking.

Finally, future work could aim to create a more lightweight deployment of the application, allowing it to run efficiently on weaker hardware and in smaller-scale environments.

Bibliography

1. Alsoussi, Abdelrahim; Tahboub, Khaldoun. Inventory management practices and challenges: an exploratory study. *An-Najah University Journal for Research-A (Natural Sciences)*. 2025.
2. Turunen, Purity. *Inventory management system for robotic labs*. Finland, 2025. Bachelor Thesis. Oulu University of Applied Sciences.
3. Brown, Simon. *The C4 model for visualising software architecture*. 2023.
4. Rumbaugh, James. *The unified modeling language reference manual*. Pearson Education India, 2005.
5. Kimball, Ralph; Ross, Margy. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

List of Figures

1.1 Sortly Items Page 11
1.2 CyberStockroom Locations Page 12
1.3 Ikea Room Customization Page 13

2.1 Flowchart for viewing the inventory list 19
2.2 Flowchart for tracking the location of a specific item 20
2.3 Flowchart for adding a new shape 22
2.4 Flowchart for editing an existing shape 23
2.5 Flowchart for removing an existing shape 24
2.6 Flowchart for adding new furniture 26
2.7 Flowchart for adding a new floor 28

3.1 System Context Diagram 30
3.2 Container Diagram 31
3.3 Component diagram for the web application(frontend) 33
3.4 Component diagram for the shape management service 34
3.5 Component diagram for the synchronization service 35
3.6 Domain model diagram 36
3.7 Mockup showing the item location page 46
3.8 Mockup showing the furniture editor page 47
3.9 Mockup showing the item list page 49

4.1 Multiselect Transfer Example 52
4.2 Action Dropdown 68
4.3 Category Creator 68
4.4 Update Item Sequence Diagram 71
4.5 Modal for Adding Polygons 76

5.1 Home Page 85
5.2 Home Page Screen 87
5.3 Shape Editor Screen 88
5.4 Selected Shape Example 88
5.5 Zone Example 89
5.6 Furniture Editor Screen 89
5.7 Furniture Definition Example 90
5.8 Floor Editor Screen 90
5.9 Floor Example 91
5.10 Item List Page 92
5.11 Shape List Page 92
5.12 View Screens 93
5.13 Item Location Examples 95

List of Tables

2.1 Mapping of functional requirements to relevant user roles 18

5.1 Keyboard and Mouse Shortcuts 88

List of Abbreviations

2D Two Dimensional. 13, 42

3D Three Dimensional. 10, 13, 42

API Application Programming Interface. 30–34, 43, 45, 50, 54, 65, 69, 70, 73, 84

BIM Building Information Modeling. 10

BIOS Basic Input/Output System. 83

CAD Computer-Aided Design. 10, 13

CPU Central Processing Unit. 83

CSS Cascading Style Sheets. 78

DFS Depth-First Search. 56

HTML HyperText Markup Language. 55, 63, 65, 66, 73, 75, 78–80, 82

HTTP Hypertext Transfer Protocol. 31

ID Identifier. 36–40, 49

IP Internet Protocol. 85

JSON JavaScript Object Notation. 31, 44, 69, 70, 81

QR Quick Response. 98

RAM Random-Access Memory. 83

REST Representational State Transfer. 50, 69, 77, 81

SQL Structured Query Language. 44, 76

UI User interface. 42, 47, 50

UML Unified Modeling Language. 19, 35

URL Uniform Resource Locator. 19, 65, 84, 85

UUID Universally Unique Identifier. 81