BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

MODULAR STUDENT CLUB MANAGEMENT SYSTEM WITH INTEGRATION INTO EXISTING INFRASTRUCTURE

MODULÁRNÍ SYSTÉM PRO SPRÁVU STUDENTSKÉHO KLUBU S INTEGRACÍ DO STÁVAJÍCÍ INFRASTRUKTURY

TERM PROJECT
SEMESTRÁLNÍ PROJEKT

AUTHOR **Bc. MAREK DANČO**
AUTOR

SUPERVISOR **Ing. ONDŘEJ ONDRYÁŠ**
VEDOUCÍ PRÁCE

BRNO 2025

Abstract

This thesis deals with the design and implementation of a modular information system composite product management, automated price calculation, tracking voluntary contributions, and offers two user interfaces, one for administration and one for sales. Backend has been implemented with ASP.NET Core and it integrates with an existing custom authentication server using Duende IdentityServer. Frontend has been implemented with React and built with Vite.

Abstrakt

Tato práce se zabývá návrhem a implementací modulárního informačního systému pro studentský klub. Obsahuje možnosti správy skladů, správy produktů a zložených produktů, automatické počítání ceny, sledování dobrovolných příspěvků a dvě uživatelská rozhraní, jedno pro administraci a druhé pro samotný prodej. Backend byl implementován v ASP.NET Core a integruje s existujícím autentizačním serverem, který používá Duende IdentityServer. Frontend byl implementován v Reactu a sestaven programem Vite.

Keywords

information system, product management, sales, modular design, web application, full-stack

Klíčová slova

informační systém, správa produktů, prodej, modulární návrh, webová aplikace, full-stack

Reference

DANČO, Marek. *Modular Student Club Management System with Integration Into Existing Infrastructure*. Brno, 2026. Term Project. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Ondryáš.

Modular Student Club Management System with Integration Into Existing Infrastructure

Declaration

I declare that I have worked on this thesis independently under the supervision of Ing. Ondřej Ondryáš. I have stated all literary sources, publications and other resources that I have used.

.....
Marek Dančo
January 12, 2026

Acknowlegments

[[**Todo**]]

Contents

List of Figures

Chapter 1

Introduction

The goal of this thesis was to study and redesign the information system used by the Kachna Student Club[1].

Any organization that handles products and their sales has a specific set of requirements for the type of products it specializes in. A lot of this is shared for all such businesses – product storage tracking, price specification and tracking, sales management and others. Systems to manage such businesses are pretty complex already and need to be secure and consistent. The Student Club however also has some specific requirements that are not easy to find in general sales systems. For example, the club functions in multiple different ways:

- Most of the time, it's open in "bar" mode, selling beer from kegs and offering a big selection of snacks.
- Once per week, it's open in "teahouse" mode, mainly preparing and selling teas and maintaining a quiet, relaxed atmosphere. During this mode, the employees are also expected to bring the orders to individual tables.
- Sometimes, it's also open for special occasions, and it serves only small subset of the usual catalogue. During these events, the orders are usually all made on the name of a single person instead of a person ordering and then paying right away as usual.

On top of this, some of the products sold are more complex than others – for example, different beer kegs might be available on different days, and it's also necessary to track the beer amounts of individual kegs.

Also, unlike most organizations, the Student Club is non-profit and requires voluntary contributions from members to keep running. Because of this, it's absolutely necessary to track the amounts of individual contributions. The contributions are also used for gamification purposes to encourage members to support the growth of the club.

Because of all these requirements, the ideal solution needs to be custom-made. In the past, there have been attempts to make such a system, but the members of the Student Union have not been satisfied with them. Some of the modules of the system are satisfactory, however, and for a new solution, it's better to integrate with them than to implement from scratch.

[1] https://su.fit.vut.cz/kachna/

Such systems are:

- KIS (Kachna Information System) Food, which handles tracking long-duration orders such as making toasts, and displaying information about them on monitors.
- KIS Auth, which handles authentication and authorization. It also integrates with the eduID[2] academic identity federation for ease of signing up.

In the following chapters, I will:

- Talk about modern information systems and what requirements are usually expected when implementing them, as well as discuss what security measures are usually employed, in chapter 2
- Familiarize the reader with the current state of the information systems used by the Kachna Student Club in chapter 3
- Analyze the requirements of the Student Union and formalize them as UML diagrams in chapter 4
- Design the individual parts of the system in chapter 5
- And finally sum up the work in chapter 6

[2] eduID.cz

Chapter 2

Modern information systems

This chapter summarizes all the typical requirements that a modern information system is expected to fulfill, that are relevant to this project. Special attention is given to security practices (2.2), as one of the requirements for this thesis was to specifically research mechanisms for authentication of users in complex information systems.

2.1 Requirements for a modern web application

These are individual characteristics of a software product that can be used to judge quality of software, taken from the ISO/IEC 25010 standard [1]. For each of the characteristics, I will either specify what is required of this project, or why the characteristic isn't very relevant to this particular software.

2.1.1 Functional suitability

The capability of the product to meet the functional needs of its users.

In the chapter 4, I will specify what exactly these needs are for this project. Some of the user requirements are mandatory, while some are only nice-to-have and might be skipped either due to lack of time or to focus on more important user needs.

2.1.2 Performance efficiency

The capability of a product to perform the functions within specified time and be efficient in the use of its resources.

A modern web application is expected to perform most of its task in time perceptible as instant by human eyes, and to provide visual feedback for the tasks that are required to take significant amount of time.

It is also expected of the system to be able to respond to all the requests the users are expected to have – for this project, however, the scalability isn't much of an issue, since the School Club is a fairly small organization with the amount of active members in dozens. The product is not expected to be deployed in a large-scale scenario, so the capacity requirement is fairly small.

2.1.3 Compatibility

The capability of a product to exchange information with other products and to perform its functions while sharing environment and resources with them.

This product is expected to work with the other modules of the Kachna Information System and to be further extended by other systems in the future.

2.1.4 Interaction capability

The capability of a product to be interacted with by users. It also includes the following:

- **appropriateness recognizability** - whether the product can be recognized by the users as appropriate for their needs,
- **learnability** - whether the product's functionality can be easily learned,
- **self-descriptiveness** - whether the product is able to make its capabilities obvious to the users,
- and **user error protection** - whether the product is able to prevent operation errors from the users.

It is absolutely necessary that the product can be recognized as appropriate. The users should be able to work with it and improve their workflow as fast as possible.

It is also very necessary to prevent operation errors, or in case of an error, helping the user resolve it as fast as possible via changing or removing the effects of the error such that the resulting state of the system correctly corresponds to the state of the real world.

Learnability and self-descriptiveness aren't as crucial to this product, as it is expected for the new operators to be schooled by the more experienced users, but it would still be very useful if the need for schooling was as small as possible, and the users were able to start using the system without too much help.

There are other parts of interaction capability that should be mentioned but aren't as relevant to this product for various reasons:

- **user engagement** - it is not necessary for the users to feel particularly engaged by the product as it is required for their work. Engagement is mostly important when users interact with the product in their free time for leisure,
- **inclusivity** - inclusivity is not very important for this particular product because it is known that only the students of FIT VUT will be using it, and as such it's not necessary to think about too wide of a spectrum of users,
- **user assistance** - it is not very important to assist users with particular needs or disabilities to the clients, because the chances of such users needing to use the product are exceedingly low. It is more important to work on other functionality of the product before considering this.

2.1.5 Reliability

The capability of a product to perform its functions without interruptions or failures.

This product should function without any failures, and it should be available under normal use. It should also be possible to recover the state of the system from an earlier point in time.

When hardware faults occur, it is acceptable for the product to stop functioning for a short period of time, but it should be able to start functioning again as fast as possible.

2.1.6 Security

The capability of a product to protect information and data, and to defend against attack patterns by malicious actors. This includes the following:

- **confidentiality** - capability to ensure that only authorized users access protected data,
- **integrity** - capability to ensure that the data cannot be modified or deleted by unauthorized users or computer error,
- **non-repudiation** - capability to prove that actions have taken place,
- **accountability** - traceability of actions within the system to a specific entity,
- **authenticity** - capability to prove that subject or resource is the one it claims to be,
- and **resistance** - capability to sustain operations even while under attack.

Confidentiality, integrity, accountability and authenticity are very important to this system, especially since it includes information about exchange of money between students and the Student Club.

Non-repudiation and resistance are still important, but not as vital, since it is not expected that anyone would want to attack the system. And while the system holds information about exchange of money, it is never expected to be treated as the source of truth, only as an accounting helper.

2.1.7 Maintainability

The capability of a product to be modified with effectiveness and efficiency. This includes:

- **modularity** - changes to one component shouldn't affect other components,
- **reusability** - capability of a product to be used in more than just one system,
- **modifiability** - capability to be modified without degrading product quality,
- and **testability** - capability to be objectively and feasibly tested for requirements.

The product should be modular, and it should be easy to change parts of it without affecting other components. It should also be easily modifiable without introducing bugs.

Reusability is not very important, as it is not planned to use the product in more scenarios than the expected scenarios.

The product should also be easily tested, and it should be easily verifiable if it serves its function correctly.

2.1.8 Flexibility

The capability of a product to be adapted to changes in requirements, contexts of use, or system environment.

Most parts of the system do not require to be deployed in different environments - the back-end and database are expected to stay the same, and the point-of-service front-end is also only expected to be used from a specific hardware. Context of use should also stay mostly the same.

The only part of the system that really needs to be flexible is the administrative web application, which should be able to run across various modern browsers, but it is again sufficient to be able to run it on desktops. Enabling mobile users to use the administrative application would be nice-to-have, but it is not a hard requirement.

2.2 Commonly used security practices

Commonly used modern security practices have shifted from simple username and password authentication to delegated authorization using OAuth 2.0 for authorization and OpenID Connect for authentication. Both of these protocols are built on top of the HTTPS protocol as a communication channel and transfer authorization data in the JSON Web Token format.

This project uses delegated authentication with OpenID Connect and it relies on the previously created subsystem "KIS Auth".

2.2.1 OAuth

OAuth 2.0 is an authorization framework that enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its behalf [2].

OAuth 2.0 is the most commonly used authorization protocol in modern web applications. It defines a way for client applications to get access to resources via **access token** - a string denoting a specific scope, lifetime and other attributes.

Access tokens should always be short-lived or single use, so for simplifying obtaining additional ones after the first sign-on, OAuth 2.0 also allows the use of a **refresh token**, which can live for longer and can be used to renew an access token.

It defines multiple flows for authorization:

- **authorization code grant**, which is used to obtain both access tokens and refresh tokens and is optimized for confidential clients,
- **implicit grant**, which is used to obtain access tokens only, and is optimized for public clients known to operate a particular redirection URI,
- **resource owner password credentials grant**, which is suitable in cases where the resource owner has a trust relationship with the client, and is used to obtain the resource owner's credentials,

- and **client credentials grant**, which is used to request an access token only using client credentials.

In modern systems, some of these flows are in the process of being deprecated. Instead, for authenticating users themselves, only the authorization code grant should be used, usually with PKCE [3] to prevent attackers from successful authorization even if they were to intercept the authorization code itself. These restrictions are currently in an active draft for OAuth 2.1 [4].

The most common OAuth 2.0 flow – authorization code flow – is depicted on the figure 2.1.

1. The client application sends its ID and redirection URI to the authorization server through its user-agent.
2. The Authorization server authenticates the user and if the authentication is successful, authorization code is returned to the client via the user-agent.
3. When the client application receives the authorization code, it sends it back to the authorization server with its redirection URI.
4. If the authorization code is valid, the authorization server responds with the access token, which can be used to access restricted resources from the resource owner, and optionally a refresh token that can be used to renew the access token.
5. While the access token is valid, client uses it to access restricted resources from the resource owner.

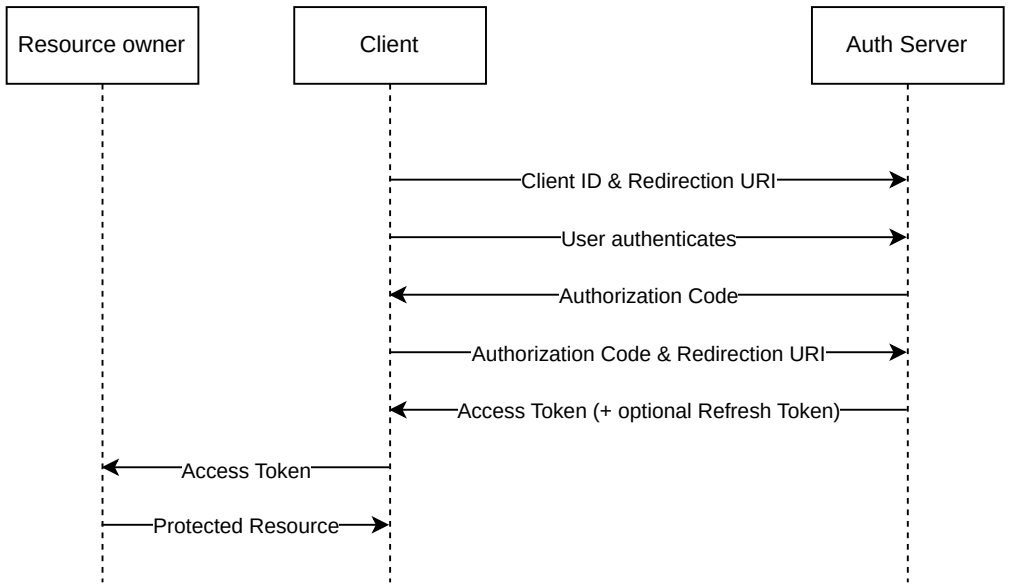

Figure 2.1: OAuth 2.0 authorization code flow

2.2.2 OpenID Connect

OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It enables clients to verify the identity of the End-User based on the authentication performed by an

Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner.

The OpenID Connect Core 1.0 specification defines the core OpenID Connect functionality: authentication built on top of OAuth 2.0 and the use of Claims to communicate information about the End-User. It also describes the security and privacy considerations for using OpenID Connect [5].

OpenID Connect is initiated when the OAuth authentication request contains the `openid` scope value. It then returns an **ID token** along with the access token from OAuth itself. The ID token contains information about the end user, such as their unique ID and optionally e-mail, name and others.

OpenID Connect also provides a way for an application to obtain user information through the **UserInfo endpoint**. Via this endpoint, a client can request additional or updated information about the end user.

2.2.3 JSON Web Token

The OAuth 2.0 protocol additionally specifies how to use bearer tokens in HTTP requests. Any party in possession of a bearer token (in OAuth, bearer is the access token) can use it to access associated resources without demonstrating possession of a cryptographic key. To prevent misuse, bearer tokens need to be protected from disclosure in storage and transport [6].

The most commonly used format of a bearer token is a **JWT** - JSON Web Token. It is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted [7].

JWTs usually consist of three distinct sections:

1. **the header**, which holds information about the type of the token and the algorithms used for signature and/or encryption,
2. **the payload**, which holds the claims about the entity represented by the JWT,
3. and **the digital signature**.

In case of unsecured tokens (which are almost never used), the algorithm in the header is set to none, and the digital signature is empty.

For usage in HTTP communications as a bearer token, each section of the JWT is represented as a UTF-8 JSON object and encoded with Base64url. Parts are then separated by the dot symbol.

Chapter 3

Current state of the information system

This chapter's purpose is to familiarize the reader with the current state of the Kachna Information System before implementing the new solutions. The current version of the information system is already in its fourth generation, but due to complex needs of the clients, a lot of needed functionality is still missing.

3.1 Overall architecture

Currently, the Kachna Information System (KIS) consists of 10 subsystems and components:

- **KIS Sales** (3.2),
- **KIS Operator** (3.3),
- **KIS Admin** (3.4),
- **KIS Auth** (3.5),
- **Kachna Online** – service for administration of club opening hours, Student Union events and user application used to access information about them,
- **KIS Monitor** – front-end application for displaying the status of longer orders,
- **KIS Food Management Device (KIS Food)** – service for publishing and management of waiting lists for longer orders,
- **KIS HW Reader** – hardware smart card reader,
- **KIS Android Reader** – smart card reader implementation for Android,
- and **KIS Reader Library** – client JavaScript library for communication with smart card readers.

The relevant subsystems will be discussed in more detail in the following sections. Interactions between individual services are illustrated in the figure 3.1.

Kachna Online, KIS Admin and KIS Operator all depend on KIS Sales, which is the service that manages the majority of the information about the system. KIS Sales in turn depends on KIS Auth for authentication, on KIS Food for scheduling and displaying longer orders, and on its database to hold the actual data.

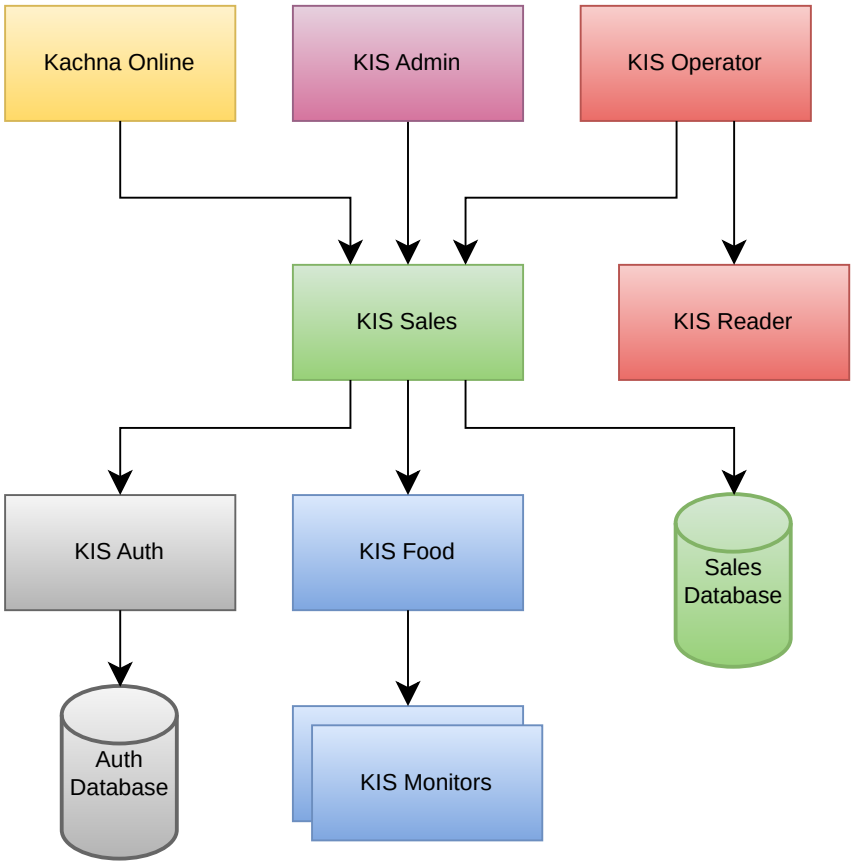

Figure 3.1: Top-level architecture of the Kachna Information System

3.2 Sales subsystem

KIS Sales is the main subsystem that the whole information system is standing on top of. It holds all the data about product storage, costs, users, voluntary contributions, beer kegs and others. The current implementation has last been updated about two and half years ago.

It is a REST application that serves as a shared back-end by Kachna Online, KIS Admin 3.4, and KIS Operator 3.3. It stores its data in a PostgreSQL database, and depends on KIS Food for queueing long preparation orders, such as toasts, and displaying them on KIS Monitors.

[1] EduId.cz

It uses Python as the main backend language, integrates directly with EduId[1] using SAML[2] authentication, and only partially integrates with the more modern authentication service KIS Auth which has also been implemented about two years ago.

The main issues stated by the Student Union with the old sales subsystem is that it doesn't provide a lot of necessary options to interact with products, such as write-offs. It also doesn't have good auditing capabilities, so when someone makes a change in the system, it is very difficult to associate the change with the user who made it.

Currently, the KIS Sales subsystem manages mainly the following entities:

- **Articles (products)** – in the current version of KIS Sales, articles are managed as simple database entities.
  Articles can also be composed of multiple different articles, but the requirement for an article to be used as component can be inconvenient – currently, only articles that can be used as components are components without set prices, components which aren't composites themselves. This can be an issue if an article can be sold separately, but can also be used as a component in a different article.
  Articles can also have any number of colored labels for filtering purposes.

- **Prices** – prices are currently statically assigned to each article, and prices of composites are not dependent on the prices of components. Because of this, when changing prices of articles, it's necessary to manually change the price of each article affected. Also, since prices are saved in a one-to-one relation with articles, it is not possible to view how the price of an article has changed over time.

- **Users** – since the previous version of the Sales system has been created before KIS Auth, the sales service is also fully capable of managing users and their data.

- **Beer kegs and taps** – in the current version of the Sales system, beer kegs are special entities that hold certain volume their assigned article. In the current schema, any article can theoretically be in a beer keg, and unsealed kegs are just identified by changes in stock that belong to them.
  An unsealed beer keg can also be opened at a beer tap. Kegs can be queried based on which tap they belong to.

- **Operations** – these include orders, contributions, stock-takings and others. Operations are core to the system, so the current implementation is one of the more mature parts of the system. However, since all the operations are grouped in one table, which results in quite messy code when it comes to handling operations. Some required operations are also not supported, such as simple write-offs of spoiled or otherwise undesirable products.

The current system has no concept of different general stores. The information about amounts of each article are stored globally, or for beer articles, associated to beer kegs which each hold only one type of article.

There also isn't an easy way to modify structure and price of a certain article for a single specific transaction. Every time a different kind of product is sold, a new special article needs to be added. This makes the database filled with products that are only slightly

[2] Security Assertion Markup Language

different from each other, like toasts with different toppings, or teas with and without milk or honey.

One more big deficiency of the current system is that it has no fixed way to display articles. All the articles are sorted alphabetically, so if a new article is added, all the following articles will get shifted and the operators constantly lose muscle memory about positions of individual articles.

3.3 Operator subsystem

Operator subsystem – KIS Operator – is the Point-of-Service web application for the bartenders that service customers during the club's opening hours. It is a front-end for the KIS Sales back-end (3.2) and offers a subset of its capabilities. It is used on specialized touch-screen devices the purpose of which is only to serve as hardware for the Operator UI.

It also integrates the KIS Reader Library for communication with a smart card reader. The reader is used to scan the students' cards for easy registration with the Student Club.

It's written in the Angular framework[3] version 12, and has last been updated approximately 3 years ago. Just quickly trying to run the application locally and installing dependencies reveals that the current implementation has over 50 security vulnerabilities registered by NPM[4], 3 of which are stated to be critical.

The main purposes of the KIS Operator currently are:

- **Communicating with the smart card reader** – this serves as the main and most convenient way for Student Club members to sign up, since all it involves is just putting their card on the reader and confirming their identity.
- **Searching normal articles** – everything except for beer from kegs should be easily searchable and possible to be added to an order in a consistent interface.
  The current KIS Operator lets the bartender search the articles, but since there isn't a clearly defined structure to the way the articles are positioned in the database, they are always just displayed in alphabetical order, which is not good for muscle memory of the bartenders.
- **Searching available beer kegs and opening them when needed** – beer kegs are special, since the club needs to track the amounts in individual opened beer kegs. Because of this, they have a separate page in the current KIS Operator.
- **Adding articles to orders** – once a particular article has been found, it needs to be added to the order in the necessary amount.
- **Submitting and cancelling orders** – for orders that have been successfully completed, it is necessary to submit them to the KIS Sales system to update the article stocks, contribution amounts for the club members and amount of cash in the used cash-box.
  For the most recent orders, it is also possible for them to be cancelled in case something went wrong or the order was made by mistake.

[3] https://angular.dev/
[4] https://www.npmjs.com/

Overall, the current KIS Operator is doing its job fairly well. The biggest problems associated with the current version is the lack of maintenance over the years, and the fact that products cannot have a fixed positioning in the Operator UI.

Some additional features would also be welcome, such as native way to handle discounts, and option to only view what articles are available in the store currently used by the bartender. These would however first need to be implemented in the Sales API, which the Operator depends on for business logic.

3.4 Administration subsystem

The administration subsystem (KIS Admin) is a web application used for administrative purposes. Similarly to KIS Operator (3.3), it is a front-end to the KIS Sales back-end (3.2).

Similarly to KIS Operator, it is written in the Angular framework version 12, and the last time it has been significantly updated is about 4 years ago. NPM also reports a high number of vulnerabilities in used libraries – 62 in total, 5 of which are critical.

The main features of KIS Admin are:

- **Article management** – browsing, creating and updating articles available for sale.
- **User management** – browsing users (Student Club members), user creation (in case it is not possible through eduID integration) and updating some of the user details, such as nickname. It is also possible to block certain card IDs from the Student Club.
- **Bar management** – browsing and creating cash-boxes and pipes.
  Browsing currently open beer kegs.
- **Operations management** – browsing and exporting all the different kinds of operations, such as orders, contributions, article stock changes, paymeents and others.
- **Report browsing** – viewing information about sales or about beer keg yields.

Different pages of the current KIS Admin UI can be seen on the figures 3.2, 3.3, and 3.4.

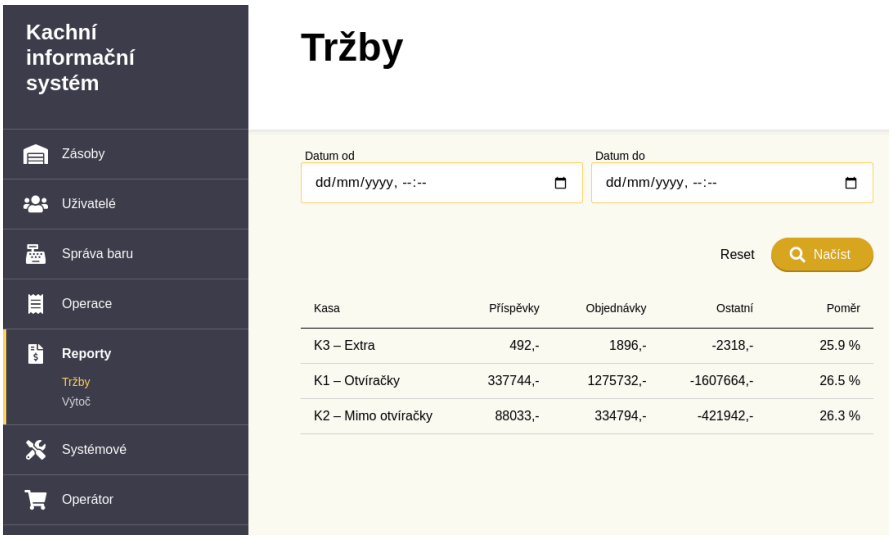

Figure 3.2: Sales report in the old KIS Admin UI

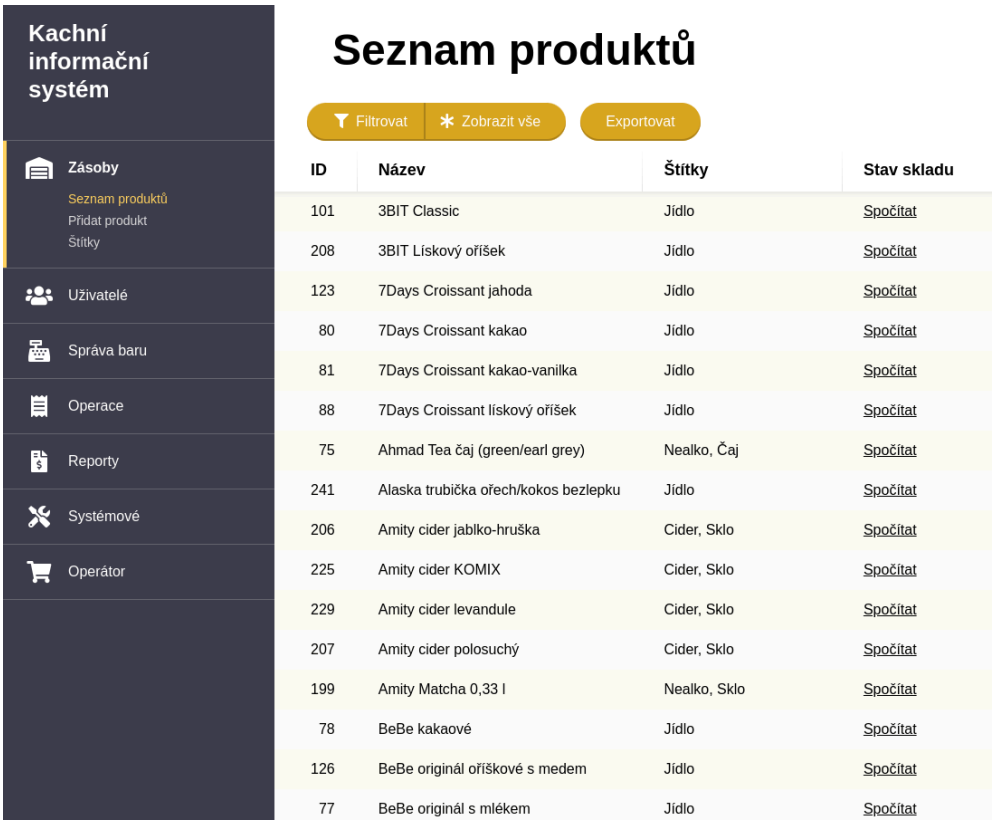

Figure 3.3: Article list view in the old KIS Admin UI

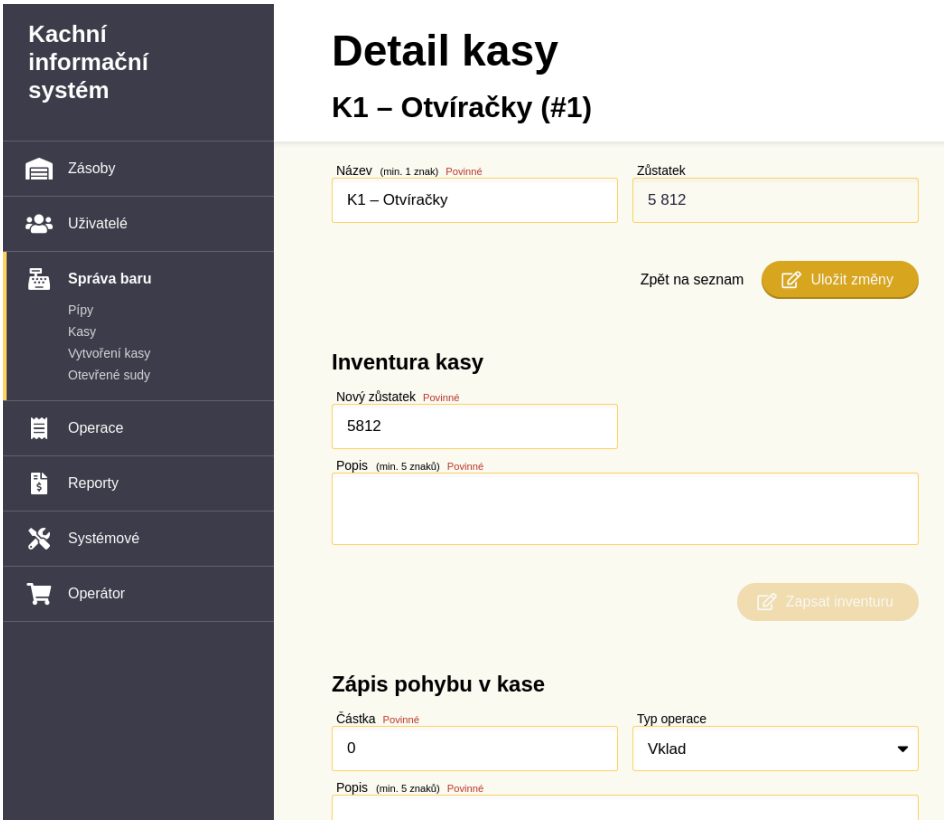

Figure 3.4: Cash-box detail view in the old KIS Admin UI

The current KIS Admin UI has several problems:

- **UI inconsistency** – data view tables on each page look slightly different, and can be interacted with in different ways. This also means updating something that should be shared between all the tables – such as pagination UI – needs to be done multiple times.
- **Lack of maintenance** – frameworks and libraries used are greatly outdated and full of vulnerabilities.
- **Lack of bulk operations** – currently, when the users want to change the amounts of multiple articles, they need to update each article individually. This can be a big problem when the users need to add a lot of different articles at once, or when amounts of products are re-counted for a stock-taking.
- **Authentication directly through Sales API** – since the KIS Admin front-end was made before the new authentication service existed, it still relies directly on KIS Sales back-end for authentication. This is not a big problem, but for modernization of the authentication, it should be integrated with the new KIS Auth (3.5) subsystem instead.

3.5 Authentication subsystem

KIS Auth is the newest addition to the Kachna Information System and handles authorization, authentication, and user management. Unlike the older system, which directly uses SAML[5] and RFID authentication on the Sales API level, KIS Auth is a separate back-end that only handles authentication. [[**Figure out what to do with RFID**]]

The new authentication system offers following ways of authenticating:

- **eduID authentication through SAML** – this was the primary way to sign in with the older KIS Sales system. It is also the only fully trusted way for user to sign up with KIS Auth, where their identity is automatically confirmed as a student.
- **RFID login through an ISIC card** – the easiest way for Club Members to log in physically at the club. This is only a way to log in, not to register. After a member has registered with a different method, they can associate a student card with their user account.
- **Discord[6] login through OAuth** – a secondary way to sign in through a less trusted provider.
- **username and password** – not often used, but still useful way for users to log in or sign up when other methods are not available.

Other services can then register as OAuth (2.2.1) clients and rely on KIS Auth to provide access tokens to authenticated users. In the new Kachna Information System, the KIS Sales back-end (3.2) should not handle authorization directly, but depend on KIS Auth.

[5] Security Assertion Markup Language
[6] discord.com

Other than just authorization through OAuth 2.0, KIS Auth also provides authentication with OpenID Connect (2.2.2) ID tokens. It is implemented in the C# programming language in .NET 8, and it uses Duende IdentityServer[7] as an implementation provider for OAuth 2.0 and OpenID Connect 1.0.

This subsystem is not currently fully integrated with the rest of the system, and one of the goals of this thesis is to integrate it with the other services. This will offer more extensibility, better separation of concerns, and more ways for users to register as club members.

[7] https://duendesoftware.com/products/identityserver

Chapter 4

Requirement analysis

This chapter describes analysis of the current requirements of the Student Union for the new, improved version of the Kachna Information System.

4.1 Informal specification

The current Kachna Information System (3) uses a solution that works, but is quite insufficient in several areas that this project is supposed to improve upon. The new system should completely replace the current implementations of the KIS Sales API and both main front-ends which depend on it – KIS Admin and KIS Operator. The new version should offer more capabilities and better integration with the new authentication system, as well as more unified UI. It would be ideal if the data was easily transferrable to the new database, but since the amount of products isn't that large, it is possible to rewrite data manually if necessary.

The general requirements are very similar to any sales management application, with some additional requirements on top. There is no requirement for the implementation platform used, but some technologies are easier to integrate with existing subsystems than others.

The full informal specification includes various levels of necessity:

- **Absolute necessities:**
  - Tracking the product amount changes over time
  - Tracking the price of individual products over time
  - Tracking individual sale transactions – how much was paid for each one and how much did the customer voluntarily contribute to the club, and at which cash-box
  - Tracking the currently open beer kegs and amounts of product in them, as well as which pipe they are opened for
- **Important features and improvements:**
  - Tracking product amounts currently available in different storage spaces

    - Auditing database changes
    - Letting customers have an "open sale transaction", where they don't pay for the product right away. This allows the bartender to update the stored amounts of products without finalizing the sale transaction completely
    - Separating products into "store items", which are bought and stored, and "sale items", which are sold and can be composed of different amounts of store items
    - Tracking sale margins of sale items and computing sale prices automatically
    - Managing "modifiers" which can be associated with different sale items to more easily alter their composition and price instead of storing every variation of a sale item separately
    - Point-of-Sales user interface with fixed positioning of sale items, where existing layouts of items don't change just by adding more items
    - Administration user interface with clean, consistent and responsive design
    - Integrating with the existing authentication system that lets students register as members of the Student Club with eduID identification (KIS Auth 3.5)
    - Integrating with the existing order-tracking system (KIS Food). This includes sending requests for queueing orders and printing the number of the order and/or number of the table for the order
- **Nice-to-haves:**
    - Managing dynamic discounts and tracking their usage by different customers. This also includes deprecating discounts that are out-of-date
    - Being able to process payments in different currencies

Some of the nice-to-have requirements may be skipped due to lack of time and prioritization of more important features.

4.2 Use-case diagram

After multiple meetings with the Student Union, the requirements have been formalized into use-case diagrams for each privileged user role in the information system. The roles are as follows:

- **Storekeeper** – manages products and their amounts in individual stores.
- **Bartender** – serves customers (student club members) during different opening modes of the student club.
- **Accountant** – makes sure that the amounts of cash in individual cash-boxes are correct and sets product prices.
- **Administrator** – manages users and all persistent entities that other roles don't have access to. Also has access to everything the other roles can do.

An use-case diagram for the given roles is depicted on the figure 4.1.

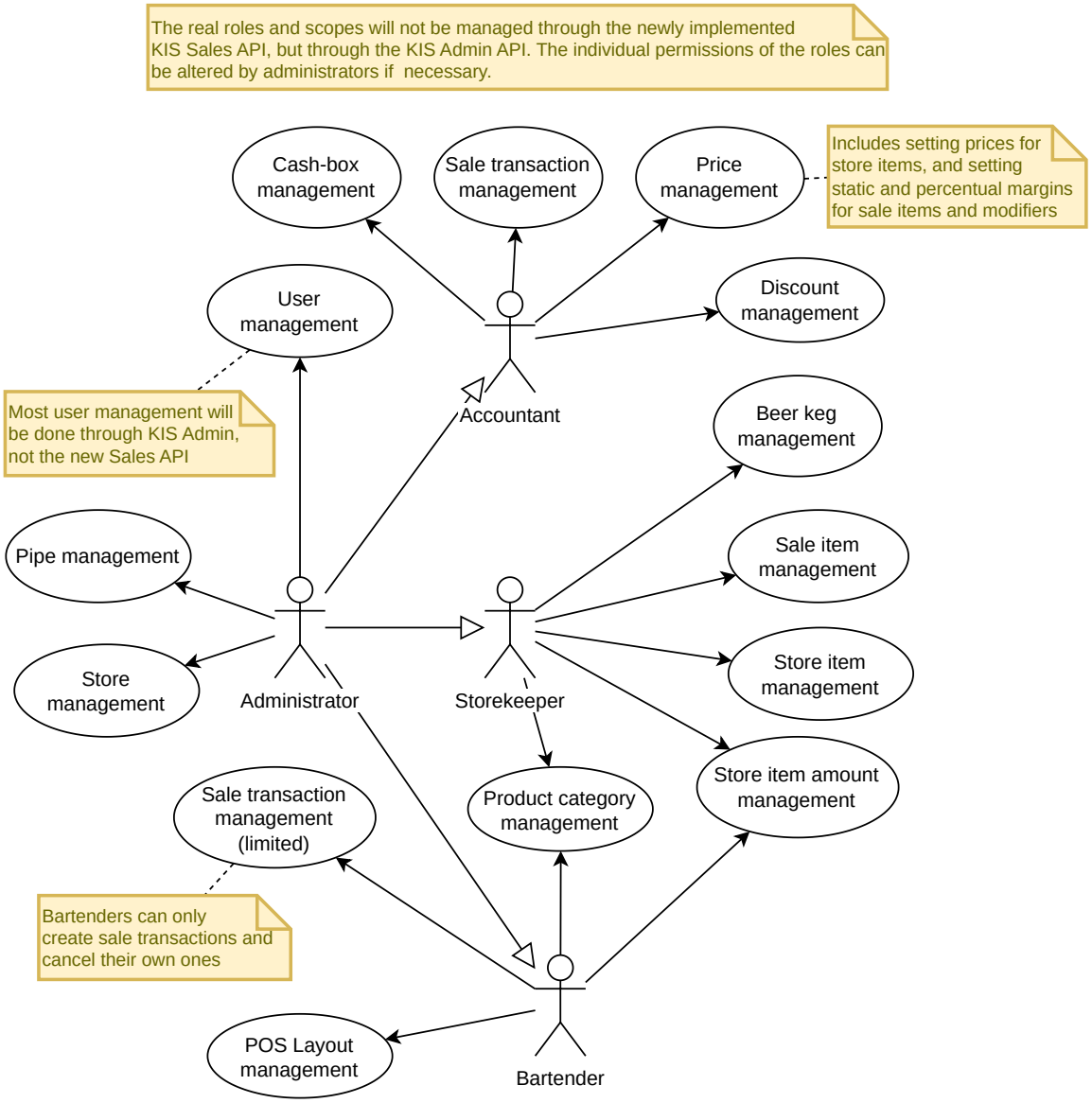

Figure 4.1: Use-case diagram for the users of the new Kachna Information System

Other than the users with privileges, basic users accounts without any privileges can also be created. These accounts represent ordinary members of the Student Club – people that don't take part in management. These users can only register, log-in, log-out and edit details of their own user account, such as their nickname in the system.

In the new version of the Kachna Information system, all of these actions will be done in the dedicated front-end of KIS Auth, not in KIS Admin like before.

4.2.1 Description of individual use cases

Store management — creating and editing store entities in which products (store items) can be stored.

Pipe management — creating and editing pipe entities that the beer kegs can be opened for.

User management — in the context of the new KIS Sales API, this will include only browsing users and displaying information about them. Other actions, such as blocking or forcefully editing other users' profiles will be provided by KIS Auth.

Cash-box management — creating and editing cash-box entities, setting the amounts of currency in each of them and displaying history of transactions for each given cash-box.

Sale transaction management — creating, updating and cancelling sale transactions. This also incudes associating sale items with sale transactions, adding modifiers to sale items, and applying discounts to sale transactions.

Price management — updating and recalculating prices for store items, changing sale margins for sale items and modifiers.

Discount management — creating and browsing discounts, displaying the usages associated with them and marking discounts as out-of-date.

Beer keg management — creating, updating and deleting beer keg types, and viewing aggregate information about them.

Sale item management — creating, updating and deleting sale items. This also includes associating sale items with their components, displaying amounts of sale items in individual stores, and associating them with categories.

Store item management — creating, updating and deleting store items. This also includes displaying amounts of store items in individual stores, and associating them with categories.

Store item amount management — adding store items to stores, moving them from one store to another, marking store items as written off and setting the amounts of store items as a stock-taking.

Product category management — creating, editing and deleting individual product categories.

POS Layout management — creating, editing and deleting fixed layouts in the Point-of-Sales UI (KIS Operator).

4.3 Data model

The figures 4.2, 4.3, 4.4 and 4.5 depict the data model of the new KIS Sales back-end, as designed according to the user requirements and some assumptions made during analysis. The diagram has been split into multiple figures because of its complexity.

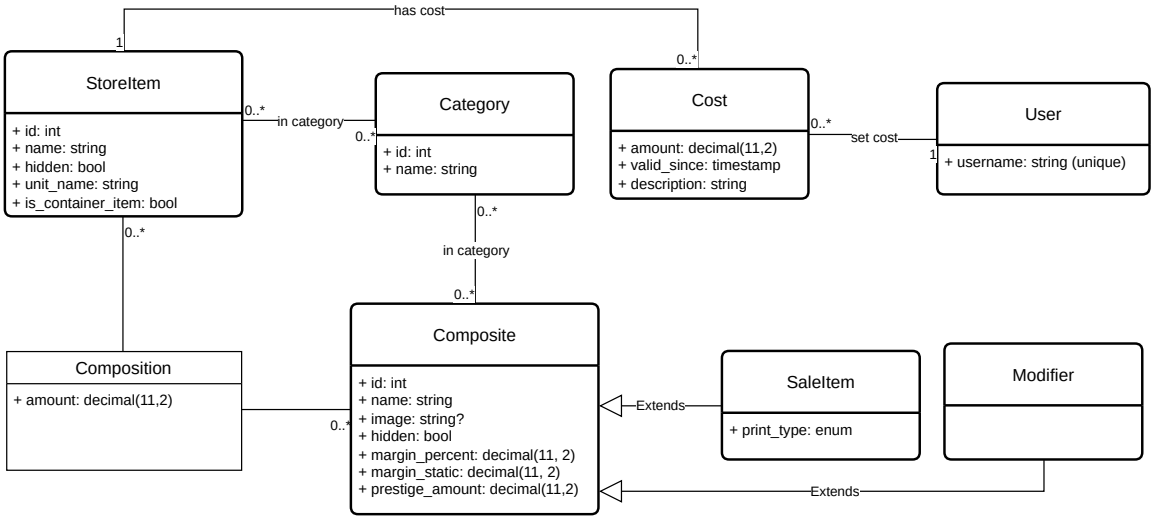

Figure 4.2: Entity Relationship diagram for the new KIS Sales – Products

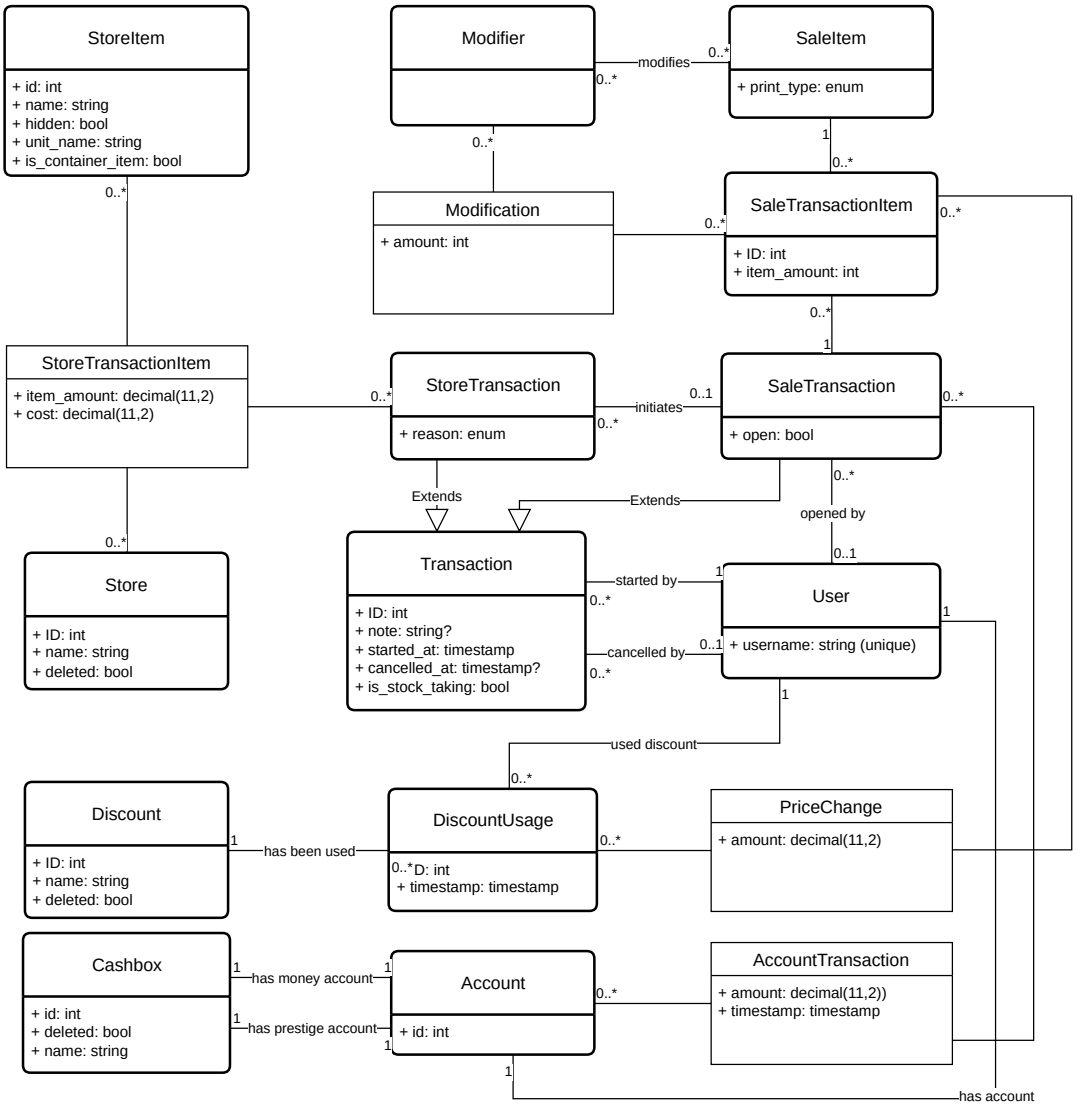

Figure 4.3: Entity Relationship diagram for the new KIS Sales – Transactions

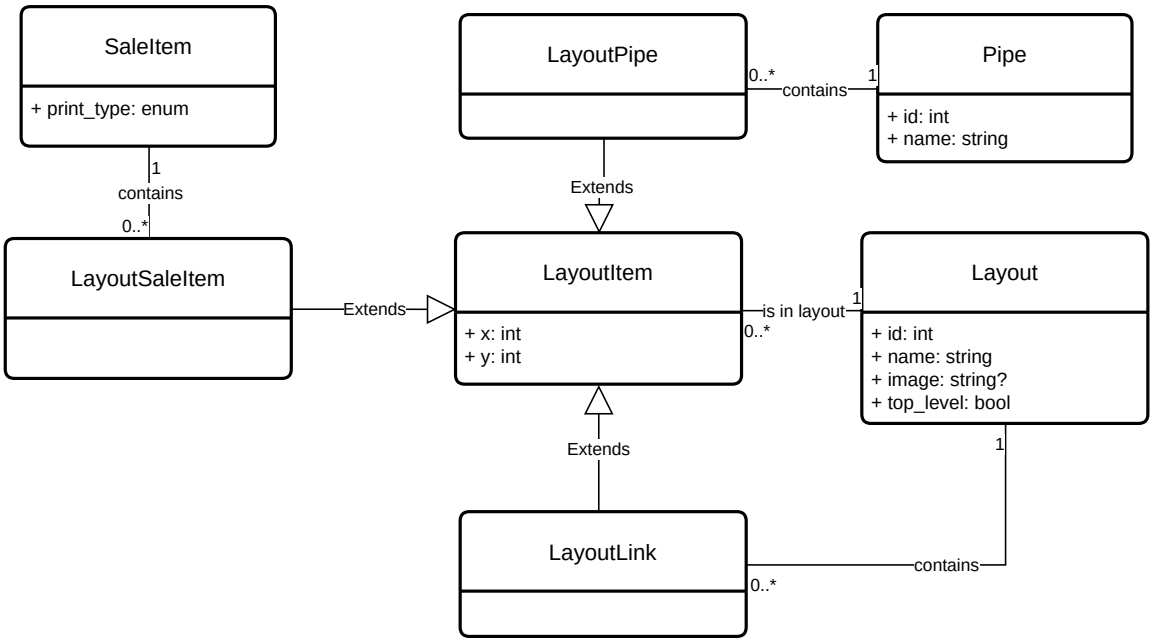

Figure 4.4: Entity Relationship diagram for the new KIS Sales – Layouts

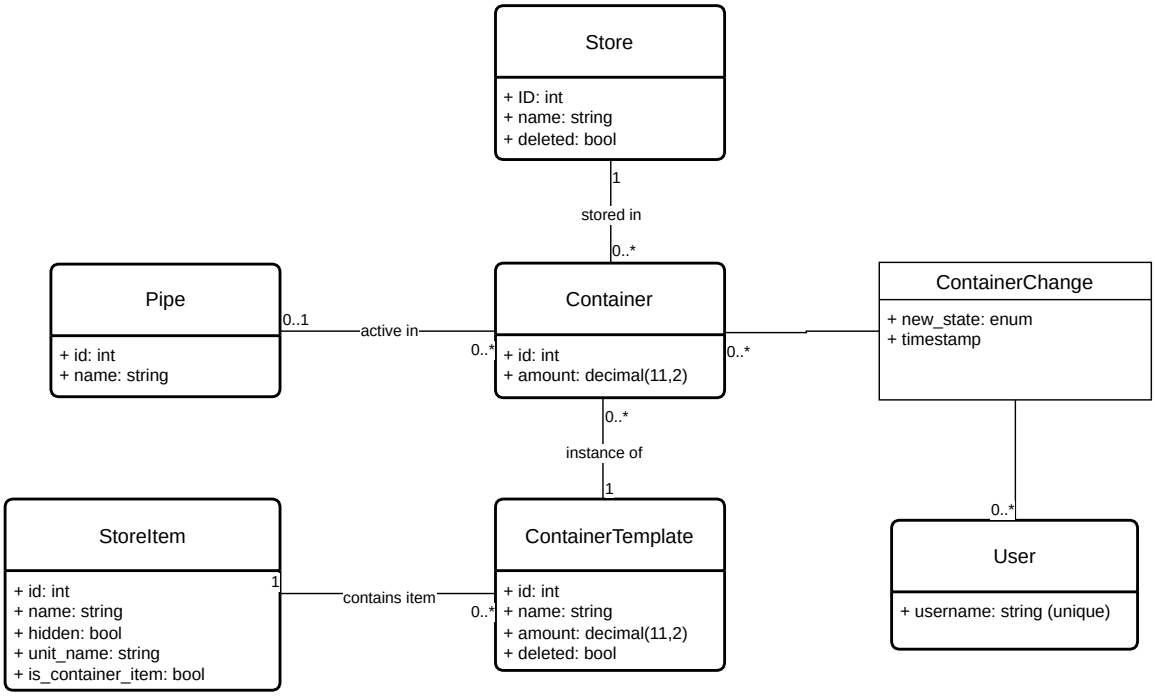

Figure 4.5: Entity Relationship diagram for the new KIS Sales – Containers (Beer kegs)

Chapter 5

Application design

This chapter describes the design of the replaced parts of the Kachna Information System, which includes the new KIS Sales back-end (3.2), KIS Operator front-end (3.3) and KIS Admin front-end (3.4).

5.1 Technology choices

Based on previous experience with web application development and the requirement analysis, the following technologies have been chosen:

- The new **KIS Sales back-end** will be implemented in **ASP.NET Core**[1] using the **C# programming language**, **Entity Framework Core ORM**[2] for interfacing with the database and **Minimal API architecture** for defining endpoints.
  - KIS Auth uses the Duende IdentityServer package to implement its authentication protocols. Documentation for this package includes many examples in C# and is made to be especially easily integrated into other C# web applications.
  - C# is also the language I have the most experience with, and it requires the least amount of familiarizing myself with the structure of applications.
  - Modern C# Minimal API architecture provides an excellent type-safe way of defining REST API endpoints that can be automatically compiled from C# code into OpenAPI documents.
  - Entity Framework Core provides a very clean and type-safe way of defining database schemas and querying for entities with speed and flexibility very close to raw SQL.
- The new **KIS Sales database** will use the **PostgreSQL RDBMS**.[3]

[1] dotnet.microsoft.com/en-us/apps/aspnet
[2] learn.microsoft.com/en-us/ef/core/
[3] postgresql.org

- PostgreSQL is the modern gold standard for relational databases because of its stability, flexibility and amount of features.

- **KIS Operator and KIS Admin** front-ends will be written in **React**[4] **with TypeScript**, with **Material UI**[5] components for UI. **Vite**[6] will be used as the build tool for optimizing front-end code and as a development server.
  - React is the most used modern web application framework and provides a convenient way to split an application into easily maintainable components.
  - TypeScript will provide type-safety and robust language server support for faster and less error-prone development compared to JavaScript.
  - Material UI is a themable and full-featured UI component framework complete with many data display components and form field inputs. It is particularly good for ensuring consistency in the user interface.
  - Vite is a unified modern web build tool which provides great developer experience with hot reload and small compile sizes in production builds.

5.2 Database design

5.3 Application architecture

5.4 User interface design

[4] react.dev
[5] mui.com
[6] vite.dev

Chapter 6

Conclusion

Bibliography

[1] *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*. 2nd ed. International Organization for Standardization, November 2023. ISO/IEC 25010:2023.

[2] Hardt, D. *The OAuth 2.0 Authorization Framework* RFC 6749. RFC Editor, January 2012. Available at: https://doi.org/10.17487/RFC6749. [cit. 2026-01-05].

[3] Sakimura, N.; Bradley, J.; Agarwal, N. *Proof Key for Code Exchange by OAuth Public Clients* RFC 7636. RFC Editor, September 2015. Available at: https://doi.org/10.17487/RFC7636. [cit. 2026-01-05].

[4] Hardt, D. *The OAuth 2.1 Authorization Framework* online. Internet Engineering Task Force, January 2025. Available at: https://datatracker.ietf.org/doc/draft-ietf-oauth-v2-1/14/. [cit. 2026-01-05].

[5] Sakimura, N.; Bradley, J.; Jones, M. B.; de Medeiros, B.; Mortimore, C. *OpenId Connect Core 1.0* online. OpenID Foundation, December 2023. Available at: https://openid.net/specs/openid-connect-core-1_0.html. [cit. 2026-01-05].

[6] Hardt, D. and Jones, M. B. *The OAuth 2.0 Authorization Framework: Bearer Token Usage* RFC 6750. RFC Editor, January 2012. Available at: https://doi.org/10.17487/RFC6750. [cit. 2026-01-05].

[7] Jones, M. B.; Bradley, J.; Sakimura, N. *JSON Web Token (JWT)* RFC 7519. RFC Editor, March 2015. Available at: https://doi.org/10.17487/RFC7519. [cit. 2026-01-05].