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 redesing of an information system for a students club. The basis of the work is an outdated information system for handling sales, product administration and user contribution monitoring. The output is a design of a more modern system, allowing also for management of multiple stores, automatic price calculation, management of product modifiers and programmable discounts. The new system is also designed to integrate with an existing authentication server.

## Abstrakt

Tato práce se zabývá úpravou návrhu informačního systému pro studentský klub. Základem práce je zastaralý informační systém pro prodej, administraci produktů a sledování členských příspěvků. Výstupem je návrh modernějšího systému umožňujícího také správu více skladů, automatické počítání cen produktů, správu modifikátorů produktů a programovatelné slevy. Nový systém je také navržen pro integraci s existujícím autentizačním serverem.

## 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 project 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 25, 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 Students Club “U Kachničky”[1]. The club functions as a meeting place for students, offers board games to borrow and play, and sells various refreshments as a non-profit organization. It is maintained by the Students Union (SU), members of which take on volunteer work as bartenders, servers, storekeepers or accountants to keep the club running.

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 Students 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 mostly tapped drinks and big selection of snacks, with volunteers working as bartenders.
- Once per week, it’s open in “teahouse” mode, mainly preparing and selling teas, and maintaining a quiet, relaxed atmosphere. During this mode, the volunteers are also expected to serve 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 in the name of a single person, instead of individual customers ordering and paying for themselves.

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

Also, unlike most organizations that deal with sales, the Students Club is non-profit and requires voluntary contributions from members to keep running. These contributions are usually taken as a part of sale transactions. Because of this, it’s absolutely necessary to track the amounts of the individual contributions within transactions. The contributions are also used for gamification purposes to encourage members to support the club.

[1]Students Club “U Kachničky”: https://su.fit.vut.cz/kachna/

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 members of the Students Union have not been satisfied with them. The implementation has been gradually growing in scale, which led to somewhat accidental architecture – it consists of multiple subsystems, some of which work together and some of which are supposed to work together but don't. In recent years, there has been an attempt to implement a better and unified authentication subsystem and integrate it, but due to the messy state of the current implementation and lack of maintainers, it wasn't fully successful.

The following chapters will contain:

- information about modern information systems and what requirements are usually expected when implementing them, as well as what security measures are usually employed (chapter 2),
- the current state of the information systems used by the Students Club "U Kachničky" (chapter 3),
- analysis of requirements of the Students Union and formalization of them into a use-case diagram (chapter 4),
- design of the individual parts of the new system (chapter 5),
- and finally the summary of the work (chapter 6).

Chapter 2

# Modern information systems

This chapter summarizes all the typical requirements that a modern information system is expected to fulfill. Special attention is given to security practices in the section 2.2, as that is one particularly important aspect of today's 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]. These characteristics will be all taken into account in the later chapters and applied for the needs of the current clients.

### 2.1.1 Functional suitability

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

- **functional completeness** – capability of a product to provide functionality that covers all the specified tasks and objectives,
- **functional correctness** – capability to provide accurate results to intended users,
- and **functional appropriateness** – capability to provide necessary and sufficient functions, and exclude unnecessary steps.

### 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. In more detail, it includes:

- **time behavior** – capability to perform functions in expected time,
- **resource utilization** – capability to not use more than required amount of resources for accomplishing functionality,
- and **capacity** – capability to function correctly under the maximum expected system load.

### 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. Its sub-characteristics include:

- **co-existence** – capability to perform functions efficiently while sharing common environment and resources with other products, while also not impacting other products' efficiency,
- **interoperability** – capability to exchange information with other products and use it,
- and **interaction capability** – capability to be interacted with by users via user interface.

### 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,
- **operability** – whether the product can be easily controlled by its intended users,
- **self-descriptiveness** – whether the product is able to make its capabilities obvious to the users,
- **user error protection** – whether the product is able to prevent operation errors from the users,
- **user engagement** – whether the product presents its functions and information in inviting and motivating manner, encouraging continued user interaction
- **inclusivity** – whether the product is usable by people of various backgrounds,
- and **user assistance** – whether the product is usable by people with different characteristics and physical and mental capabilities.

### 2.1.5 Reliability

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

- **faultlessness** – capability to function without fault under normal operation,
- **availability** – capability to be operational when required for use,
- **fault tolerance** – capability to operate despite the presence of faults in the system,
- and **recoverability** – capability to recover the data from the system and continue work after interruption and major failure.

### 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.

### 2.1.7 Maintainability

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

- **modularity** – capability of individual components to keep functioning in the same way even after updating 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.

### 2.1.8 Flexibility

The capability of the product to be adapted to changes in requirements, contexts of use or system environment. This includes:

- **adaptability** – capability to be adapted for different hardware, software or other environments,
- **scalability** – capability to handle growing or shrinking workloads while maintaining similar efficiency,
- **installability** – capability to be installed and/or uninstalled the required environments,
- **replaceability** – capability to replace a different product for the same purpose in the same environment.

## 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 usually transfer authorization data in the JSON Web Token format.

### 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 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.

Multiple protocol flows are defined within OAuth 2.0 [2] 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.
2. The Authorization server authenticates the user and if the authentication is successful, authorization code is returned to the client.
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.

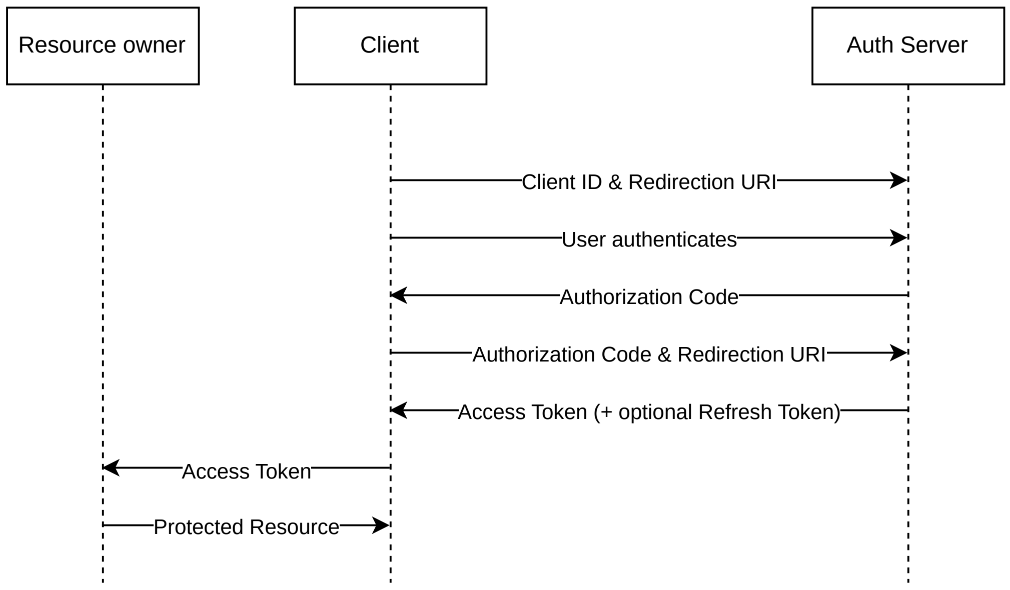

Resource owner
Client
Auth Server
Client ID & Redirection URI
User authenticates
Authorization Code
Authorization Code & Redirection URI
Access Token (+ optional Refresh Token)
Access Token
Protected Resource

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 [5].

The OpenID Connect Core 1.0 specification [5] 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.

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 is a data structure containing claims about the authentication of an end user by an authorization server, and it is represented as JSON Web Token (described in Subsection 2.2.3) containing claims about the token and the end user. The minimum contained claims are:

- `iss` – the issuer of the token (URL of the authentication server),
- `sub` – the subject of the token (end user identifier),
- `aud` – audience or audiences of the token, represented as either a client ID or array of client IDs,

- `exp` – the expiration timestamp of the token, represented as a Unix timestamp in seconds,
- and `iat` – the timestamp at which the token was issued, represented as a Unix timestamp in seconds.

The ID token can contain other standard claims, such as authentication time of the end user, nonce, as well as other custom claims specific to the used authentication server.

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, which are normally represented by a JSON object that contains a collection of name and value pairs for the claims.

### 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 [6].

The most commonly used format of a bearer token is the JSON Web Token – **JWT**, described in [7]. 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.

JWTs 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 encoding. 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 state of the currently employed system, as a whole called "Kachní informační systém", or KIS. The current version of the information system is already in its third generation, but due to complex needs of the clients, a lot of needed functionality is still missing. Also, due to lack of maintenance, there is a lot of architecture and security problems.

## 3.1 Overall architecture

Currently, KIS consists of 8 loosely interconnected subsystems and components:

- **KIS Sales** (further described in Section 3.2) – the main "source of truth" REST API which holds all the core information about the system and business logic,
- **KIS Operator** (further described in Section 3.3) – front-end to the KIS Sales API used by volunteers in the Students Club as a Point-of-Sales system,
- **KIS Admin** (further described in Section 3.4) – another front-end to the KIS Sales API used for the administration of the Students Club,
- **KIS Auth** (further described in Section 3.5) – new authentication system that isn't fully integrated with the other systems, but offers unified authentication for all the users of KIS,
- **Kachna Online** – service for administration and displaying of club opening hours and Students Union events,
- **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,

- and **KIS Reader** – hardware smart card reader used for RFID identification of club members.

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 relies on KIS Food for scheduling and displaying longer orders, and for KIS Auth to authenticate against KIS Food.

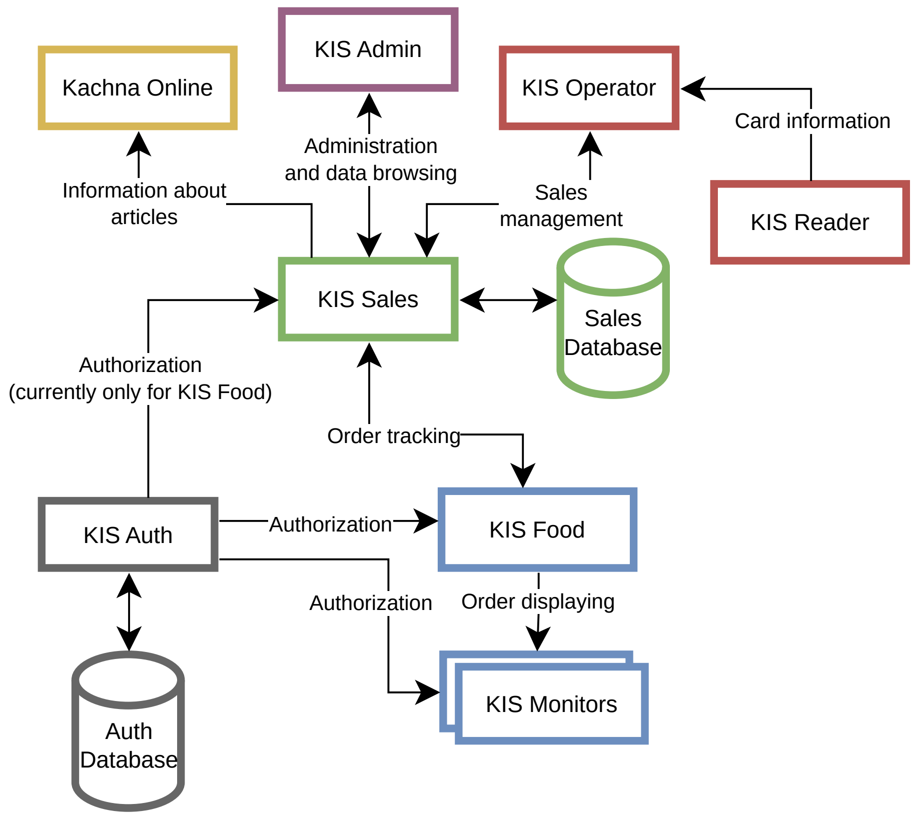


Figure 3.1: Top-level architecture of the currently employed information system for the Student Club “U Kachničky”. The arrows show the flow of data and the colors show very strongly connected components.

## 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, 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 (Section 3.4), and KIS Operator (Section 3.3). It stores its data in a PostgreSQL[1]

[1]https://www.postgresql.org/

database, and depends on KIS Food (Section 3.6) for queueing long preparation orders, such as toasts, and displaying them on KIS Monitors.

It uses Python as the main backend language, integrates directly with EduId[2] using SAML[3] authentication, and only partially integrates with the more modern authentication service KIS Auth which has also been implemented about two years ago. It uses its own authentication for most of the operations it supports, but it relies on KIS Auth to authenticate against the newer KIS Food subsystem.

The main issues stated by the Students 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, the only articles that can be used as components are articles without set prices and articles 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 – components can't even have a set price. Because of this, it's necessary to always manually set the price of each article that requires a new price every time the price changes. 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.

- **Kegs and taps** – in the current version of the Sales system, kegs are special entities that hold certain volume their assigned article. In the current schema, any article can theoretically be in a keg, and unsealed kegs are just identified by changes in stock that belong to them. An unsealed keg can also be opened at a 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, all the operations are grouped in one table, which results in quite messy code when it comes to handling them. Some required operations are also not supported, such as simple write-offs of spoiled or otherwise undesirable products.

[2]https://www.eduid.cz

[3]Security Assertion Markup Language: https://wiki.oasis-open.org/security/

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

There also isn't an easy way to modify the structure and price of a certain article for a single specific transaction. Every time a different kind of product starts being sold, a new special article needs to be added. This fills the database with products that are only slightly different from each other, like toasts with different toppings, or teas with and without milk or honey.

## 3.3 Operator subsystem

Operator subsystem – KIS Operator – is the Point-of-Sales 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 (Section 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 contribution tracking.

It's written in the Angular framework[4] 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[5], 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 Students Club members to log in, since all it involves is just putting their card on the reader and confirming their identity. The cards are also used to track contributions of each member, as all members must scan their card before completing every transaction.
- **Searching normal articles** – everything except for drinks 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 browse the articles and filter them based on labels, 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 kegs and opening them when needed** – kegs are special, since the club needs to track the amounts in individual opened 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.

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

- **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.

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, option to only view what articles are available in the store currently used by the bartender, and option to modify articles for individual transactions. These would however first need to be implemented in the Sales API, which the Operator depends on for business logic.

The main KIS Operator UI can be seen on the Figure 3.2, and the UI for displaying taps and kegs on the Figure 3.3.

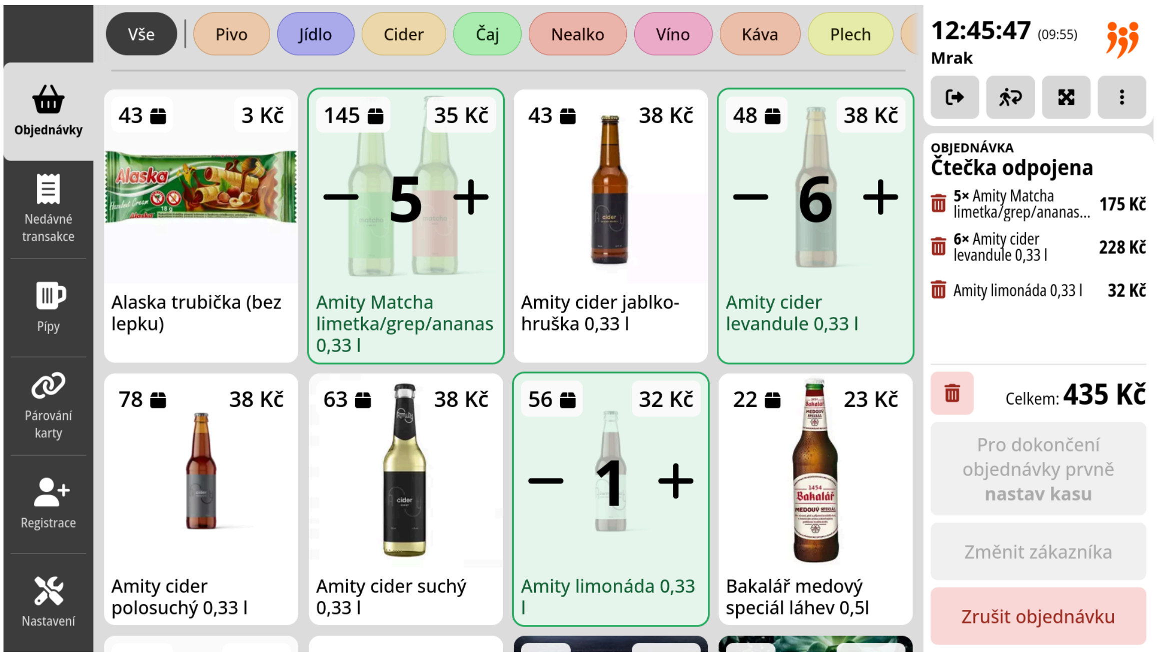

Vše
Pivo
Jídlo
Cider
Čaj
Nealko
Víno
Káva
Plech
12:45:47 (09:55)
Mrak
Objednávky
Nedávné transakce
Pípy
Párování karty
Registrace
Nastavení
43 3 Kč
Alaska trubička (bez lepku)
145 35 Kč
– 5 +
Amity Matcha limetka/grep/ananas 0,33 l
43 38 Kč
Amity cider jablko-hruška 0,33 l
48 38 Kč
– 6 +
Amity cider levandule 0,33 l
78 38 Kč
Amity cider polosuchý 0,33 l
63 38 Kč
Amity cider suchý 0,33 l
56 32 Kč
– 1 +
Amity limonáda 0,33 l
22 23 Kč
Bakalář medový speciál láhev 0,5l
OBJEDNÁVKA
Čtečka odpojena
5× Amity Matcha limetka/grep/ananas... 175 Kč
6× Amity cider levandule 0,33 l 228 Kč
Amity limonáda 0,33 l 32 Kč
Celkem: 435 Kč
Pro dokončení objednávky prvně nastav kasu
Změnit zákazníka
Zrušit objednávku

Figure 3.2: Main user interface in the current version of KIS Operator. It allows displaying the individual articles with their current stock amounts, filtering of the articles by labels and managing the amounts of articles in the currently open order.

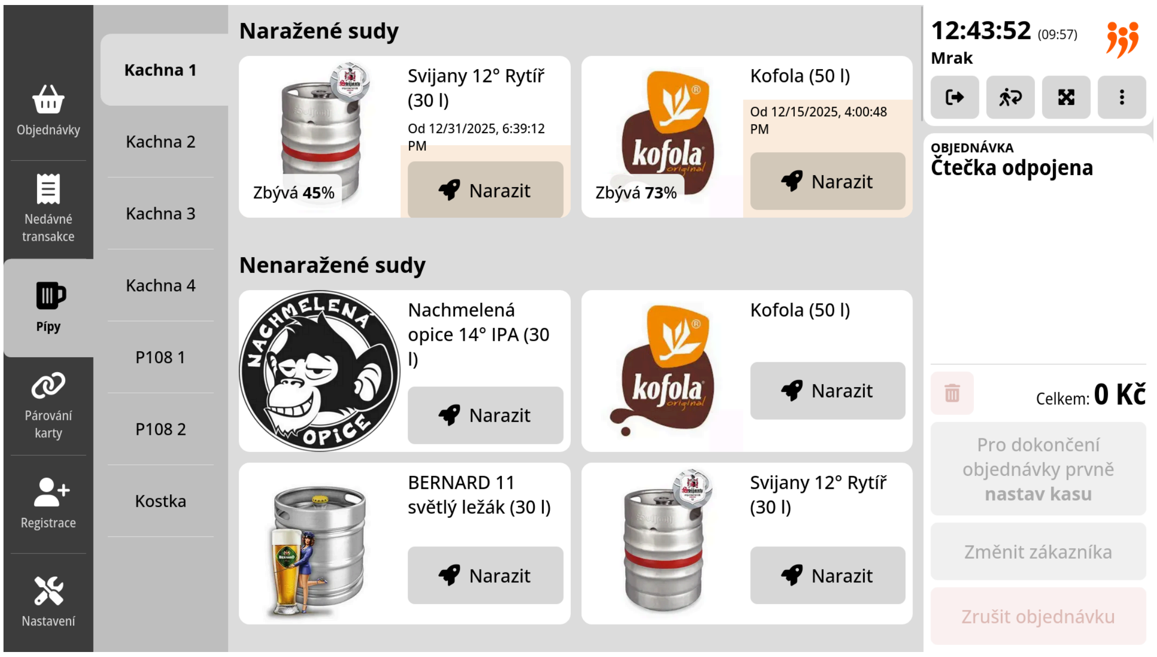


Figure 3.3: The user interface for managing taps in the current version of KIS Operator. It allows displaying the currently available kegs, their current approximate amounts, and opening them when needed.

## 3.4 Administration subsystem

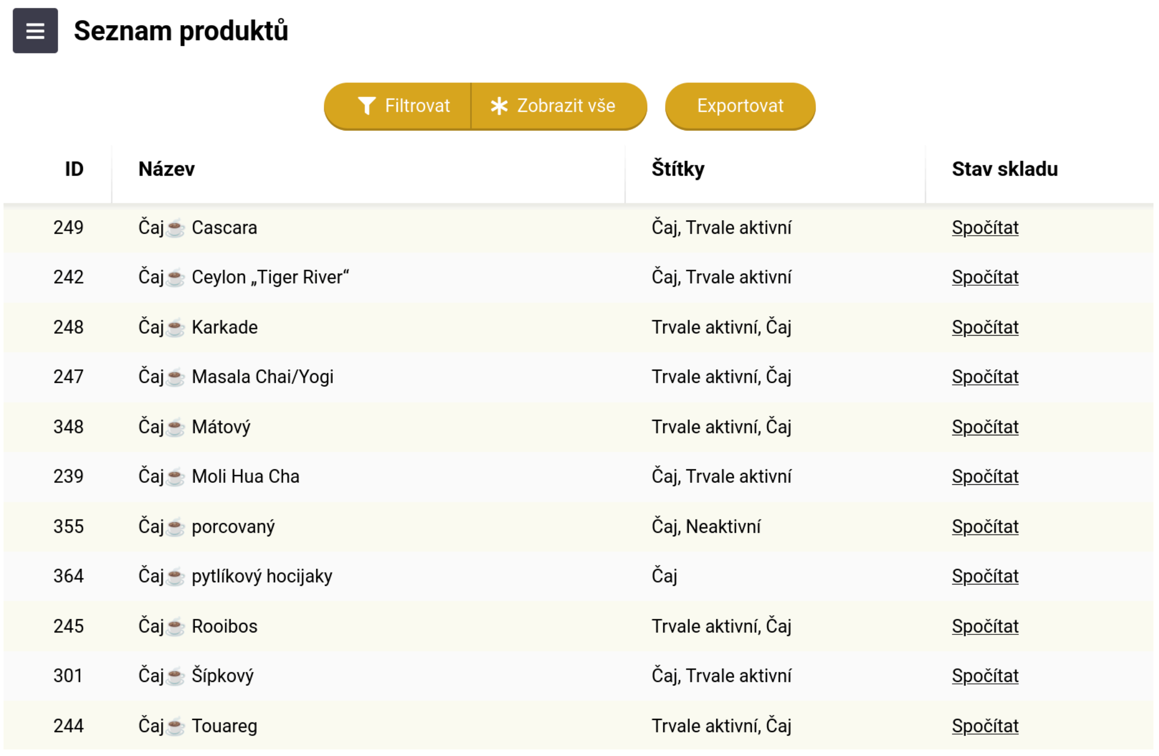
The administration subsystem (KIS Admin) is a web application used for administrative purposes. Similarly to KIS Operator (Section 3.3), it is a front-end to the KIS Sales back-end (Section 3.2).

Like 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:

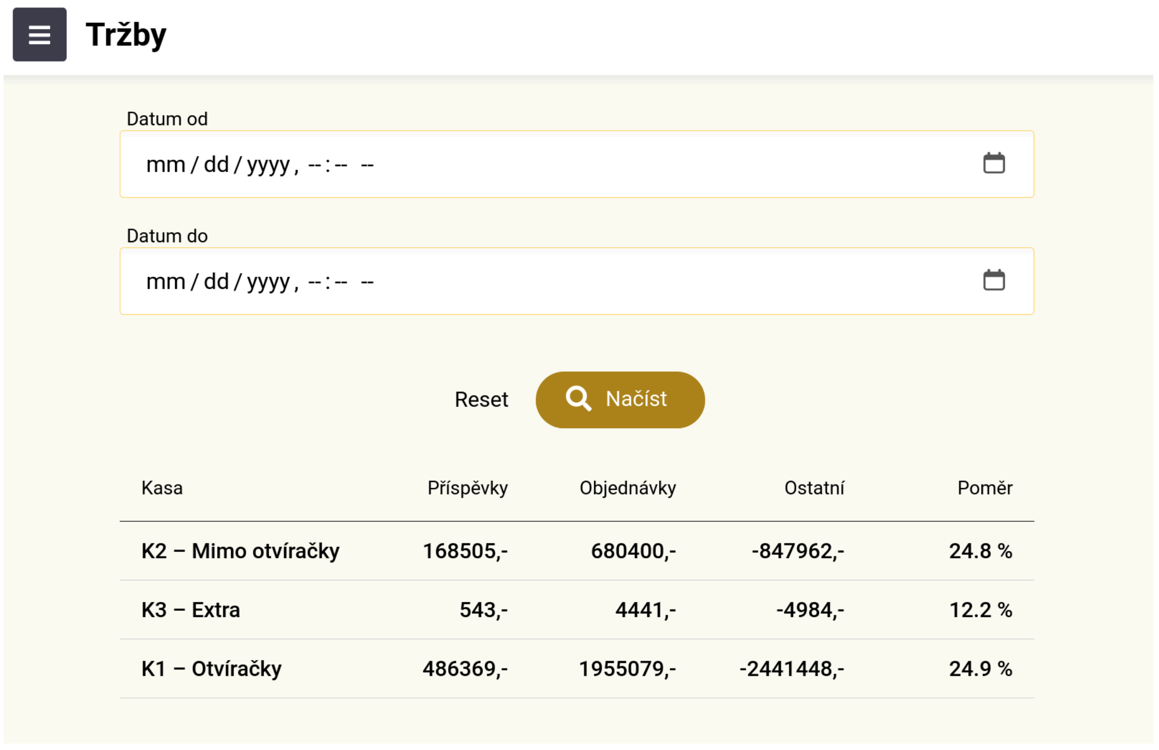
- **Article management** – browsing, creating and updating articles available for sale.
- **User management** – browsing users (Students Club members), user creation (in case it is not possible through eduID integration) and updating some of the user details, such as nickname. It is also possible to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

Different pages of the current KIS Admin UI can be seen on the Figures 3.4, 3.5 and 3.6.


Seznam produktů

| ID | Název | Štítky | Stav skladu |
|---|---|---|---|
| 249 | Čaj☕ Cascara | Čaj, Trvale aktivní | Spočítat |
| 242 | Čaj☕ Ceylon „Tiger River“ | Čaj, Trvale aktivní | Spočítat |
| 248 | Čaj☕ Karkade | Trvale aktivní, Čaj | Spočítat |
| 247 | Čaj☕ Masala Chai/Yogi | Trvale aktivní, Čaj | Spočítat |
| 348 | Čaj☕ Mátový | Trvale aktivní, Čaj | Spočítat |
| 239 | Čaj☕ Moli Hua Cha | Čaj, Trvale aktivní | Spočítat |
| 355 | Čaj☕ porcovaný | Čaj, Neaktivní | Spočítat |
| 364 | Čaj☕ pytlíkový hocijaky | Čaj | Spočítat |
| 245 | Čaj☕ Rooibos | Trvale aktivní, Čaj | Spočítat |
| 301 | Čaj☕ Šípkový | Čaj, Trvale aktivní | Spočítat |
| 244 | Čaj☕ Touareg | Trvale aktivní, Čaj | Spočítat |

Figure 3.4: Article list view in the old KIS Admin user interface – it displays each article, its labels and optionally the current stock status.

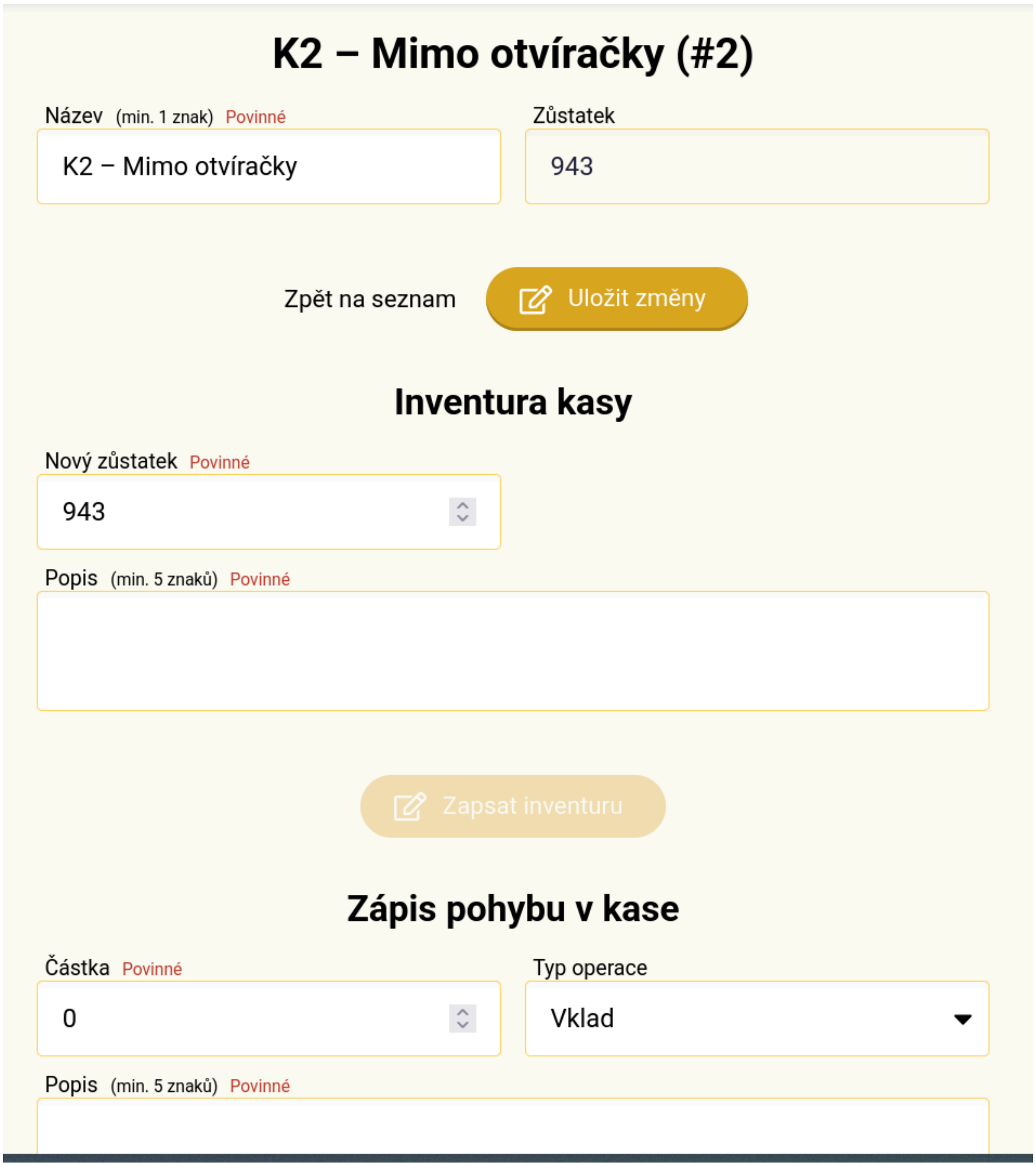

Tržby

Datum od
mm / dd / yyyy , -- : -- --

Datum do
mm / dd / yyyy , -- : -- --

| Kasa | Příspěvky | Objednávky | Ostatní | Poměr |
|---|---|---|---|---|
| K2 – Mimo otvíračky | 168505,- | 680400,- | -847962,- | 24.8 % |
| K3 – Extra | 543,- | 4441,- | -4984,- | 12.2 % |
| K1 – Otvíračky | 486369,- | 1955079,- | -2441448,- | 24.9 % |

Figure 3.5: Sales report in the old KIS Admin user interface – it displays the total amounts of contributions, order payments, and other cash movements for each cash-box.


K2 – Mimo otvíračky (#2)

| | |
|---|---|
| Název (min. 1 znak) Povinné | Zůstatek |
| K2 – Mimo otvíračky | 943 |

Inventura kasy

| | |
|---|---|
| Nový zůstatek Povinné | 943 |
| Popis (min. 5 znaků) Povinné | |

Zápis pohybu v kase

| | |
|---|---|
| Částka Povinné | Typ operace |
| 0 | Vklad |

Popis (min. 5 znaků) Povinné

Figure 3.6: Detail view of a cash-box in teh old KIS Admin user interface – it displays the current amount of cash in the cash-box, and lets the users change the cash-box name, save a cash change, or perform a stock-taking.

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 (Section 3.5) subsystem instead.

## 3.5 Authentication subsystem

KIS Auth is the newest addition to KIS and handles authorization, authentication, and user management. Unlike the older system, which directly uses SAML[6] and RFID[7] authentication on the Sales API level, KIS Auth is a separate back-end that only handles authentication.

It is implemented in the C# programming language in .NET 8, and it uses Duende IdentityServer[8] as an implementation provider for OAuth 2.0 and OpenID Connect 1.0.

KIS Auth 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. With other approaches, the identity has to be confirmed by an existing account with corresponding privileges.
- **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 their student card with their user account.
- **Discord[9] login through OAuth** – a secondary way to sign up 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. This way, an account is directly created in KIS Auth without using any external services like all the other methods.

Other services can then register as OAuth clients (Subsection 2.2.1) and rely on KIS Auth to provide access tokens to authenticated users. Other than just authorization through OAuth 2.0, KIS Auth also provides authentication with OpenID Connect (Subsection 2.2.2).

[6]Security Assertion Markup Language: https://wiki.oasis-open.org/security/

[7]Radio Frequency Identification – contactless short-distance identification using smart cards. The Student Union uses MIFARE chips embedded into Czech ISIC cards: https://www.mifare.net/

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

[9]https://discord.com/

This subsystem is not currently fully integrated with the rest of the system – it is only used for authenticating the KIS Sales back-end against the order tracking subsystem (described in section 3.6). Integrating it fully would make the whole system more extensible, have better separation of concerns, and let club members to sign up in more different ways.

## 3.6 Order tracking subsystem

The order tracking subsystem – KIS Food – is a back-end service that handles order queueing requests from the KIS Sales API. It automates the workflow of tracking, creating and handing out numbered tickets for individual long-term orders by the volunteers working at the bar.

Similarly to KIS Auth (Section 3.5), it is implemented in the C# language using .NET 8. Communication between individual devices managed by KIS Food back-end is performed with the SignalR library[10].

KIS Food subsystem also includes:

- monitoring devices that show the state of individual orders and call out finished orders via text-to-speech,
- and management devices used to print paper slips with numbers and barcodes, and to scan the barcodes to change the state of an order.

The main KIS Food API then serves as a communication channel between the KIS Sales back-end, order monitoring devices, and management devices. HTTP requests are sent to KIS Food, which then communicates its state with monitors and management devices.

[10]SignalR – real-time web communication library: https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-8.0

# Chapter 4

# Requirement analysis

This chapter describes analysis of the current requirements of the Students Union for the new, improved version of KIS.

## 4.1 Informal specification

The current KIS (chapter 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 kegs and amounts of product in them, as well as which taps 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 Students Club with eduID identification (KIS Auth – Section 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

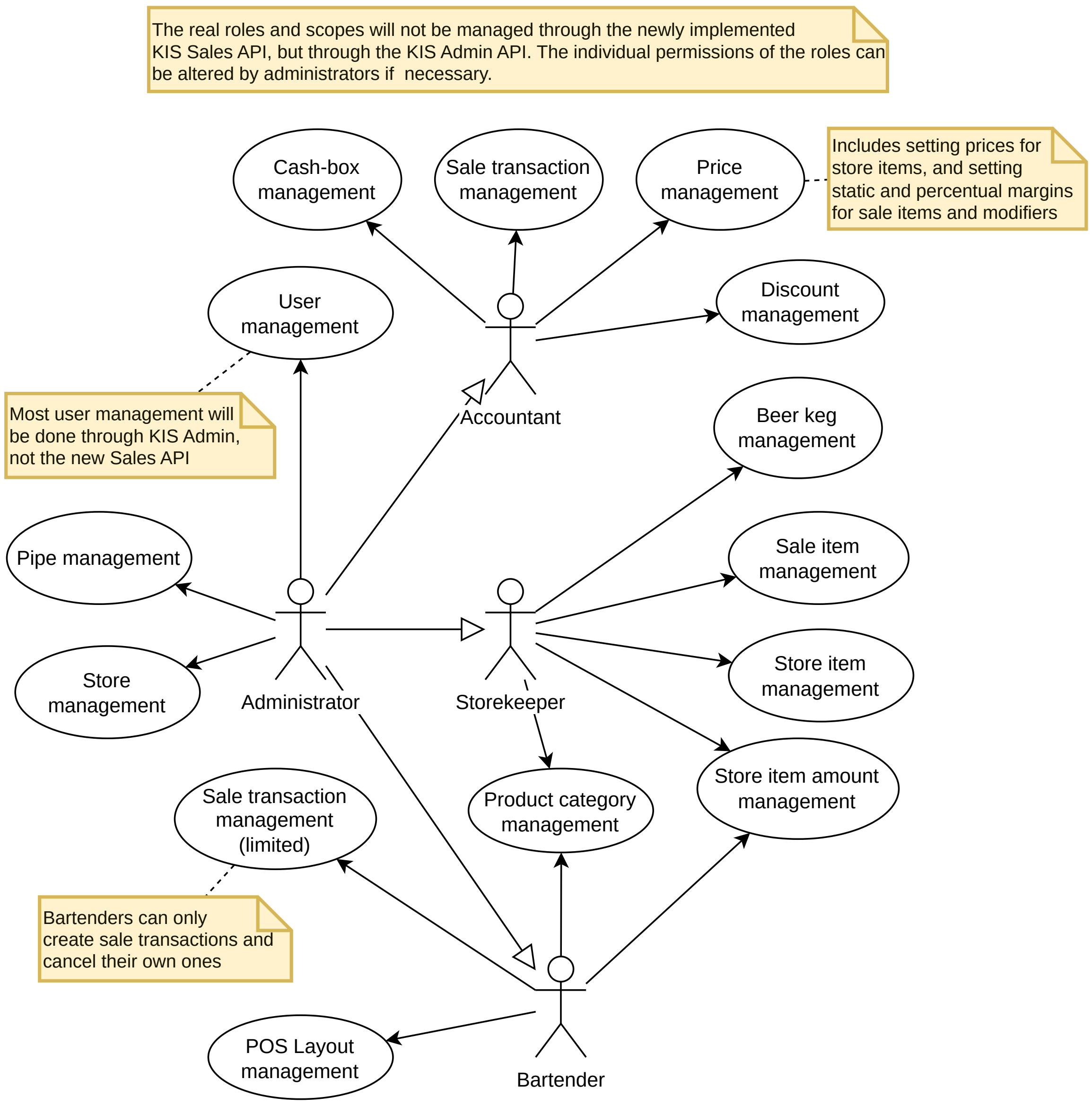
Some of the nice-to-have requirements may be skipped due to lack of time and prioritization of more important features.

## 4.2 Use-case diagram

After multiple meetings with the Students 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.



The real roles and scopes will not be managed through the newly implemented KIS Sales API, but through the KIS Admin API. The individual permissions of the roles can be altered by administrators if necessary.
Cash-box management
Sale transaction management
Price management
Includes setting prices for store items, and setting static and percentual margins for sale items and modifiers
User management
Discount management
Accountant
Most user management will be done through KIS Admin, not the new Sales API
Beer keg management
Pipe management
Sale item management
Store management
Administrator
Storekeeper
Store item management
Store item amount management
Sale transaction management (limited)
Product category management
Bartenders can only create sale transactions and cancel their own ones
POS Layout management
Bartender

Figure 4.1: Use-case diagram for the users of the 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 Students 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.

### 4.2.1 Description of individual use cases

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

**Tap management** — creating and editing tap entities that the 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.

**Keg management** — creating, updating and deleting 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

Based on the informal specification and the use-case diagram, a data model has been created to include all the necessary entities for proper functionality of the system. The appendix A shows the full entity relationship diagram. The main entities tracked by the new system are:

- **Store items** – items that are bought by the club and stored in the individual stores.
- **Costs** – buying costs for store items. The active cost for a given store item is always the last set one, and the prices for composite items are computed from the active costs of its store items.

- **Stores** – places where the individual store items or kegs can be stored in. The store items are stored by creating store transaction items which track the amounts of added or removed amounts of store items within the store.
- **Containers** – represent individual kegs, or potentially other container entities that could be used in the future. Their state is also tracked, and state changes are timestamped.
- **Container templates** – added to the new system to simplify the creation and tracking of individual keg types.
- **Taps** – represent taps where the kegs can be opened.
- **Categories** – an equivalent to labels in the older system, can be used to sort store items or composite items.
- **Composites** – items that are composed of any number of different store items in specific amounts. To compute their prices, they store sales margin in percentage and fixed value.
  - **Sale items** – composites which can be sold as individual items.
  - **Modifiers** – composites which can be used to modify the prices and compositions of sale items.
- **Transactions** – similar to operations in the older system. Transactions represent changes in the store item amounts in the individual stores or changes in account currency amounts. Transactions can also be marked as stock-takings, which don't represent changes in the system, but rather setting the amounts of items to their real values in the store.
  - **Store transactions** – represent changes of store item amounts in the specified stores. Each store transaction also tracks the reason for why it was performed, for example addition of items, moving of items or write-offs.
  - **Sale transactions** – represent the sale of sale items along with specified modifiers. Each sale transaction can also have multiple store transactions associated with it. It is also tracked how much each sale transaction changed the amounts of cash in a cash-box and for a user.
- **Users** – user accounts are stored to track individual club members' contribution, but also to track the operations in the database. For every important action, such as changing a price of a store item, starting or cancelling a transaction, and discount usage, the responsible user is tracked.
- **Cash-boxes** – represent real-life cash-boxes and track the amounts of cash and contributions in them.
- **Layouts** – store information about how the individual layout items should be displayed in the Point-of-Sales UI. Each layout represents a grid and holds a number of layout items, which can be sale items, taps, or links to other layouts.

- **Discounts** – represent the discounts that could be implemented by the Student Union. Each discount also tracks its usages for each user, and how much each discount usage changes the price of the affected sale items.

After talking in more detail with the Students Union, it has been decided that the requirement to process payments in multiple currencies would require a lot of effort and is not very important compared to other requirements. Because of this, the option to pay in multiple currencies has been removed for this project, and it might be implemented as an extension in the future.

# Chapter 5

# Application design

This chapter describes the design of the replaced parts of KIS, which includes the new KIS Sales back-end (Section 3.2), KIS Operator front-end (Section 3.3) and KIS Admin front-end (Section 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 the **C# programming language**.
    - KIS Auth uses the Duende IdentityServer[1] 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 written in it.
- The new **KIS Sales database** will use the **PostgreSQL RDBMS**[2].
    - PostgreSQL is the modern gold standard for relational databases because of its stability, flexibility and amount of features. According to the DB-Engines ranking [8], it is the second most popular open-source database in the world.
- **KIS Operator and KIS Admin** front-ends will be written in **React**[3] **with TypeScript**. **Vite**[4] will be used as the build tool for optimizing front-end code and as a development server.

[1]https://duendesoftware.com/products/identityserver
[2]https://www.postgresql.org/
[3]https://react.dev
[4]https://vite.dev/

- React is the most used modern web application framework (according to [9]) and provides a convenient way to split an application into easily maintainable components.
- TypeScript will provide type-safety and a robust language server support[5] for less error-prone development compared to JavaScript.
- Vite is a unified modern web build tool which provides great developer experience with hot reload and small compile sizes in production builds. It is rapidly growing in popularity, being currently the second most used after WebPack, according to [9].

## 5.2 System architecture

A redesign of an information system also includes redesigning the architecture. The architecture of the older system, described in Section 3.1, was not ideal, as the new authentication subsystem, which was supposed to provide authorization and authentication to all the other systems, was not fully integrated.

One more small issue with the older architecture was the fact that KIS Admin and KIS Operator do not necessarily need to remain separate. According to the new use-case diagram, shown in figure 4.1, bartenders and storekeepers share some responsibilities. Because of this, it's better to reuse the same components for both KIS Admin and KIS Operator.

In the new system, KIS Operator will be added as a component to the KIS Admin. They will share the same application, which will also simplify the authorization, since instead of two separate clients, only one will need to be configured.

The new architecture is depicted on the figure 5.1.

## 5.3 Database design

The database was designed first, as the whole structure of the system depends on the structure of its data. All the required entities from the data model (Section 4.3) need to be represented well for the use-case as possible. The final database design closely resembles the entity relationship diagram (Appendix A).

### 5.3.1 Denormalization

Some denormalizations have been implemented to simplify and speed up the read queries on the database. In some instances, it is better to store the same data in multiple different locations to improve performance.

For all the entities that have state tracked by other entities (for example store item costs), it is practical to store the tracked state at least partially in the entity itself. This will

[5]TypeScript Language Server: `https://github.com/typescript-language-server/typescript-language-server`

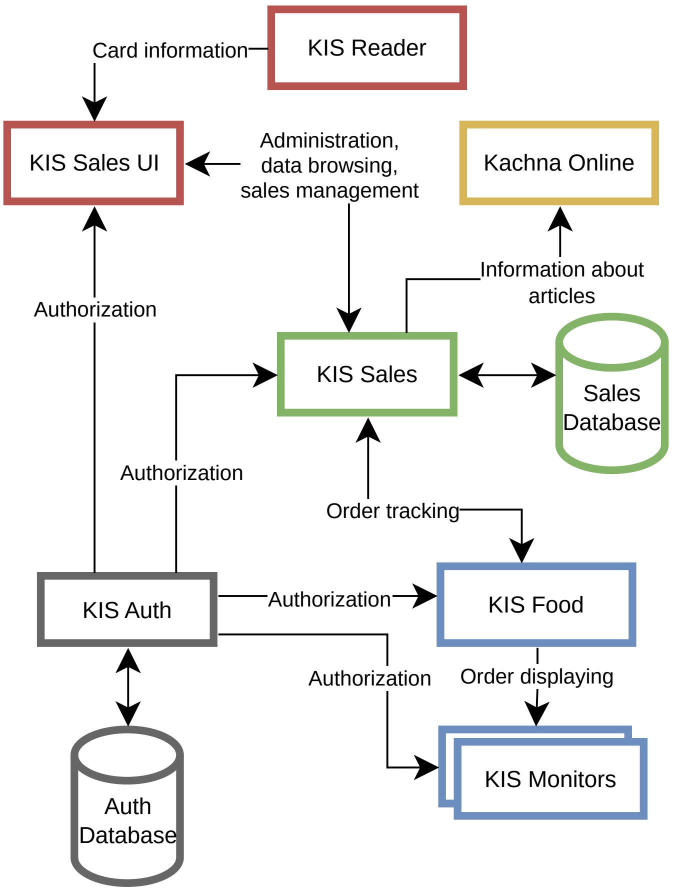


Figure 5.1: Top-level architecture of the new information system for Students Club “U Kachničky”. The arrows show the flow of data and the colors show very strongly connected components.

simplify the queries and speed them up due to smaller number of joins needed. Example of this can be seen on the Figure 5.2.

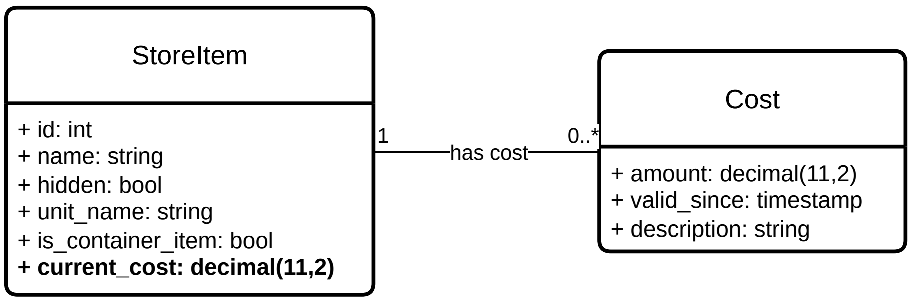


Figure 5.2: Part of an ER diagram showing denormalization of the store item costs.

Another useful denormalization is tracking the amounts of store items and composites in the individual stores. This information is necessary to be as recent as possible, so the bartenders know whether it is possible to sell a given product in a given amount. However, aggregating the amounts of items every time would be costly and inefficient. For this reason, it is more practical to create tables to track these amounts. This can be seen on the Figure 5.3.

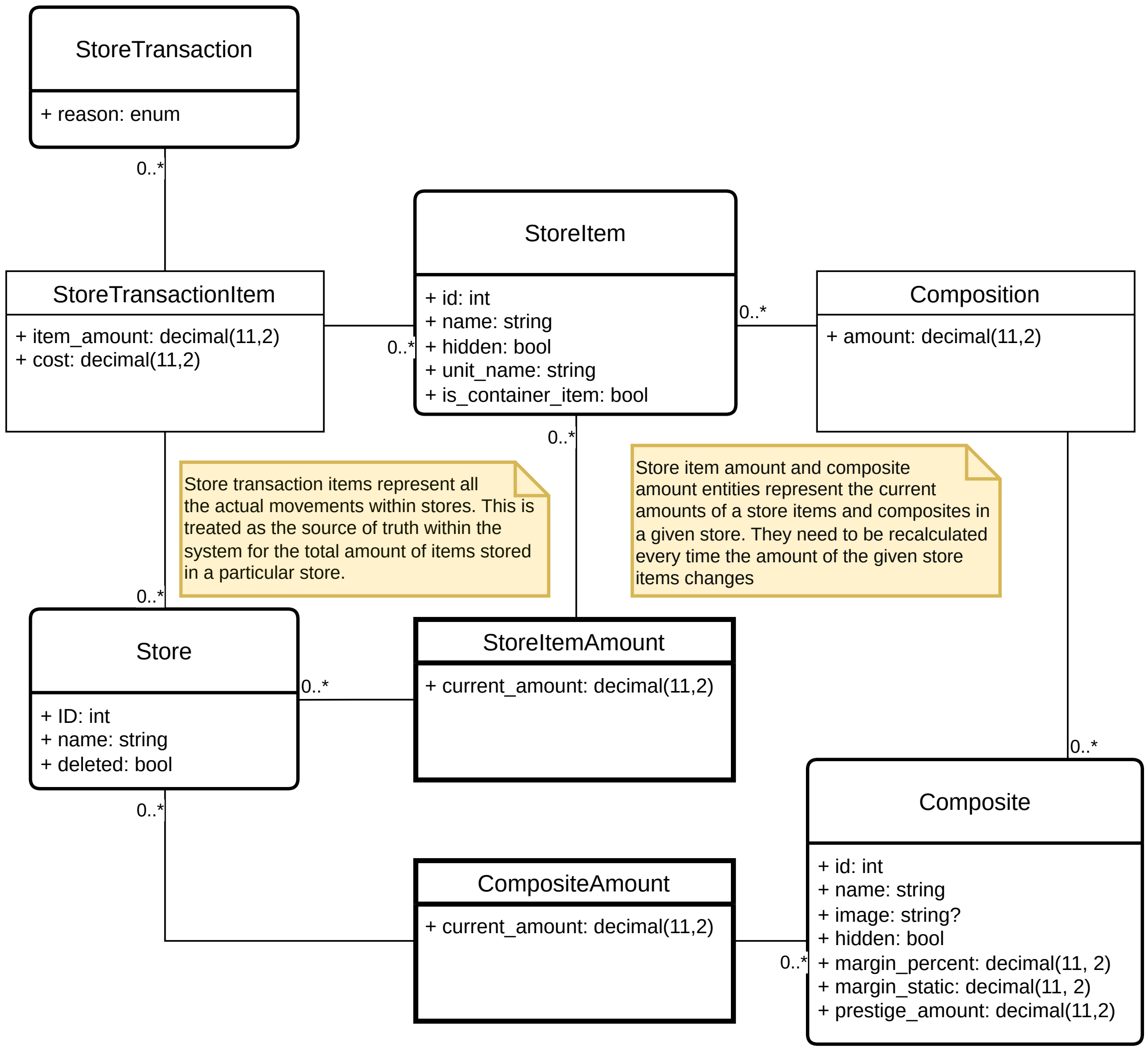

StoreTransaction
+ reason: enum
0..*
StoreItem
+ id: int
+ name: string
+ hidden: bool
+ unit_name: string
+ is_container_item: bool
StoreTransactionItem
+ item_amount: decimal(11,2)
+ cost: decimal(11,2)
0..*
0..*
Composition
+ amount: decimal(11,2)
0..*
Store transaction items represent all the actual movements within stores. This is treated as the source of truth within the system for the total amount of items stored in a particular store.
Store item amount and composite amount entities represent the current amounts of a store items and composites in a given store. They need to be recalculated every time the amount of the given store items changes
0..*
Store
+ ID: int
+ name: string
+ deleted: bool
0..*
StoreItemAmount
+ current_amount: decimal(11,2)
0..*
Composite
+ id: int
+ name: string
+ image: string?
+ hidden: bool
+ margin_percent: decimal(11, 2)
+ margin_static: decimal(11, 2)
+ prestige_amount: decimal(11,2)
0..*
CompositeAmount
+ current_amount: decimal(11,2)
0..*

Figure 5.3: Path of an ER diagram showing denormalization of store item amounts and composite amounts within stores.

## 5.3.2 Inheritance

When implementing a relational database with inheritance, it is necessary to decide what inheritance pattern to use to best handle the data. In the KIS Sales database, there are three inheritance hierarchies – composites, layout items and transactions. For each of these hierarchies, the vast majority of the data is shared in the parent type, and in case of layout items, all the data is shared. The different types of layout items hold the same data, the only difference is that the foreign keys for each derived type points to a different table.

For tables where the majority or all the data is shared between derived types, it is best to use the table-per-hierarchy pattern. In this pattern, all the types are stored in one table and to distinguish between them, a discriminator field is added. This approach potentially takes up more space than others, but it is the simplest to query, as no joins are needed.

### 5.3.3 Object-relational mapping

With modern systems, one of the common decisions is whether to use an ORM system or not. In most cases, it is highly beneficial to use it, as it provides safer way to query data, easier and type-safe data definition, and ease of switching the underlying database in case it's necessary.

In the minority of cases, the database communication needs to be extremely fast, and it's necessary to optimize the queries to the utmost limit. However, this project will be deployed on a very small scale with only few people using it at the same time. All the benefits of an object-relational mapping system will be felt very strongly when defining the data model in code, while the main benefit of using SQL directly – performance – would not be felt very much.

Because of this, the ORM Entity Framework Core[6] has been chosen as the way to communicate with the database. Entity Framework Core is the official ORM directly from Microsoft, and it allows creating code-first data model and sync it with a database using migrations. It also supports querying data in a type-safe way with LINQ, the alternative to SQL built into C#.

## 5.4 Back-end REST API

The REST API is the backbone of the system, and as such, it needs to be as robust and maintainable as possible. The .NET ecosystem offers an official framework for building web applications that is open-source, actively maintained and supported by Microsoft – ASP.NET Core[7]. It is the standard for developing modern web applications, so it was an easy choice as the main framework to use.

### 5.4.1 Endpoint definition

ASP.NET Core offers two approaches of defining REST API endpoints:

- **Controllers** – an older way that defines endpoints as methods on "controller" classes. With this approach, most metadata about endpoints is set via attributes. It is a time-proven and supports a wide range of use-cases. The structure of the API with this approach is fixed, which simplifies the design, but can sometimes feel like "magic", where a lot of information is just inferred from conventions and non-type safe attributes.
- **Minimal APIs** – the more modern approach where methods are directly mapped to specific endpoints using type-safe syntax. It doesn't give the developers clear direction and conventions like controller-based approach does, but that also means more flexibility. The main advantage of this approach is its type-safety – it is not necessary to specify return types of methods, HTTP verbs, or any other metadata with attributes, as everything is defined via the type system and extension methods. Minimal APIs

[6]https://learn.microsoft.com/en-us/ef/core/
[7]https://dotnet.microsoft.com/en-us/apps/aspnet

are also significantly faster than controllers, and they're currently recommended by Microsoft as the default for new web applications.

For this project, Minimal API approach has been chosen for its type-safety, which enables very easy definition of OpenAPI[8] documents. With the OpenAPI description, it is possible to easily generate front-end code for interacting with the API itself, which simplifies the workflow of developing a full-stack application. The speed of Minimal APIs is also a welcome feature, while not being as important as the type-safety and ease of use.

### 5.4.2 Architecture

APIs handle multiple different responsibilities – database access, business logic, and data validation. To keep them maintainable and modular, it is necessary to split the code into different layers, where each layer has a strict responsibility. In this application, the following structure has been chosen:

1. **Database access layer** – this layer defines the structure of the database, individual entities, and connections between them. It also defines global entity filters (for entities that are stored in the database, but are not supposed to be visible in standard queries), structure of the inheritnace hierarchies, and conventions, such as data formats.

2. **Business layer** – this layer is responsible for interacting with the data access layer and performing all the business logic. It deals with selecting and updating the correct entities, handling database transactions, and performing complex aggregations when needed. This is the most complex layer

## 5.5 User interface design

[8]https://www.openapis.org/

Chapter 6

# Conclusion

The main goal of this project was to redesign an information system tailored to the real-world needs of a specific students club. The new system will manage product stock around multiple stores, sales, student contributions, static product layouts, individual kegs, and dynamic discounts. It will provide database auditing, product cost tracking and automatic price calculation. It will also integrate fully with the newer authentication subsystem through OAuth 2.0 and OpenID Connect.

Compared to the older system, the newer system will offer more functionality that has been missing until now. It will also remove a lot of security vulnerabilities caused by unmaintained libraries.

General improvements include:

- Back-end: More modern, layered .NET architecture with type-safety, better OpenAPI document generation and thorough validation. Instead of a few very complex files that each define complete business logic, the project will be more organized and easier to maintain.
- Front-end: More consistent UI thanks to using a component library, automatically generated SDK for communication with the back-end from the OpenAPI document, and Point-of-Sales UI that stays consistently ordered when new products are added.
- Security: Integration with existing infrastructure that was mostly unused until recently

# Bibliography

[1] ISO/IEC JTC 1/SC 7 Technical Commitee *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].

[8] Red Gate Software *DB-Engines Ranking* online. 2025. Available at: https://db-engines.com/en/ranking. [cit. 2026-01-19].

[9] Devographics *State of JavaScript* online. 2024. Available at: https://2024.stateofjs.com/en-US/. [cit. 2026-01-19].

## Appendix A

# Entity relationship diagram

The Figures A.1, A.2, A.3 and A.4 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.

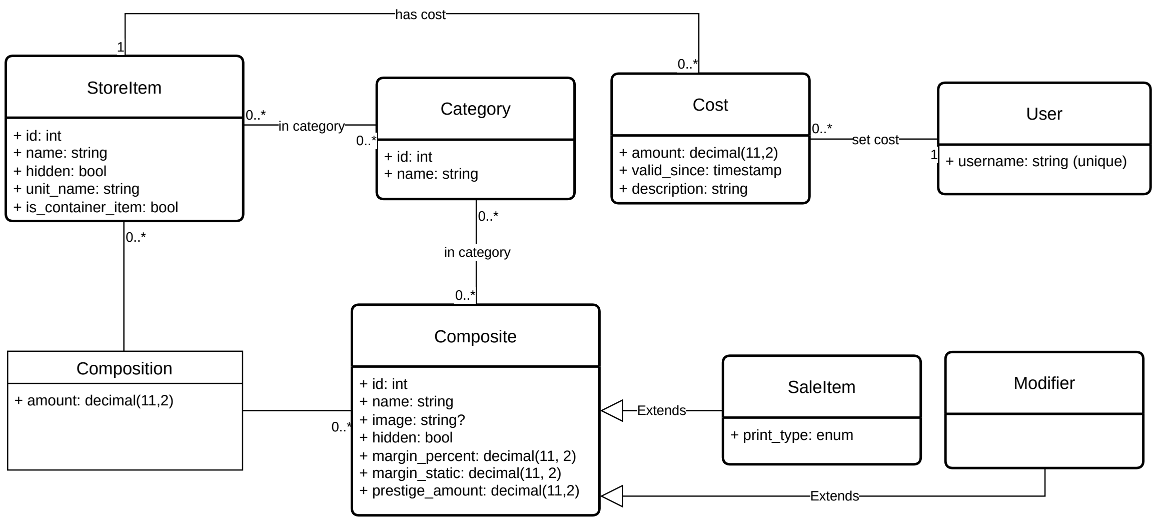


Figure A.1: Entity Relationship diagram for the new KIS Sales – Products

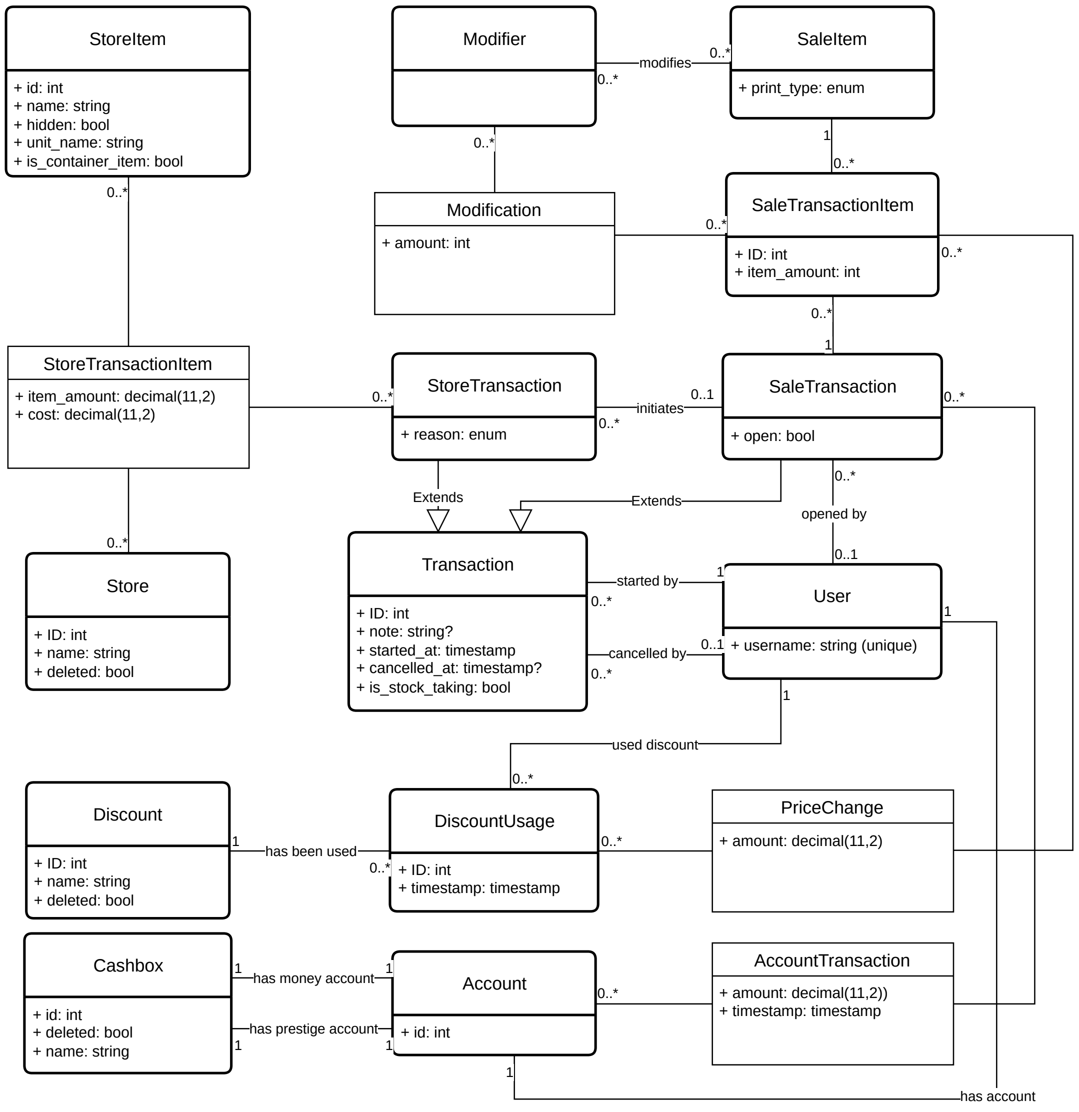


Figure A.2: Entity Relationship diagram for the new KIS Sales – Transactions

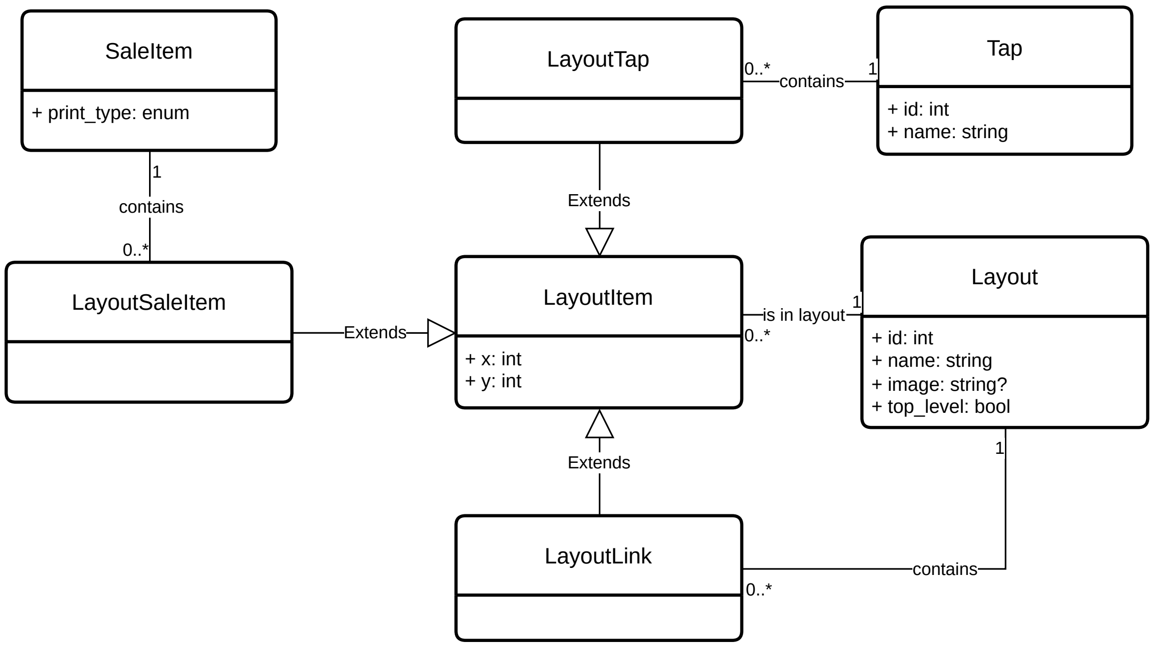


Figure A.3: Entity Relationship diagram for the new KIS Sales – Layouts

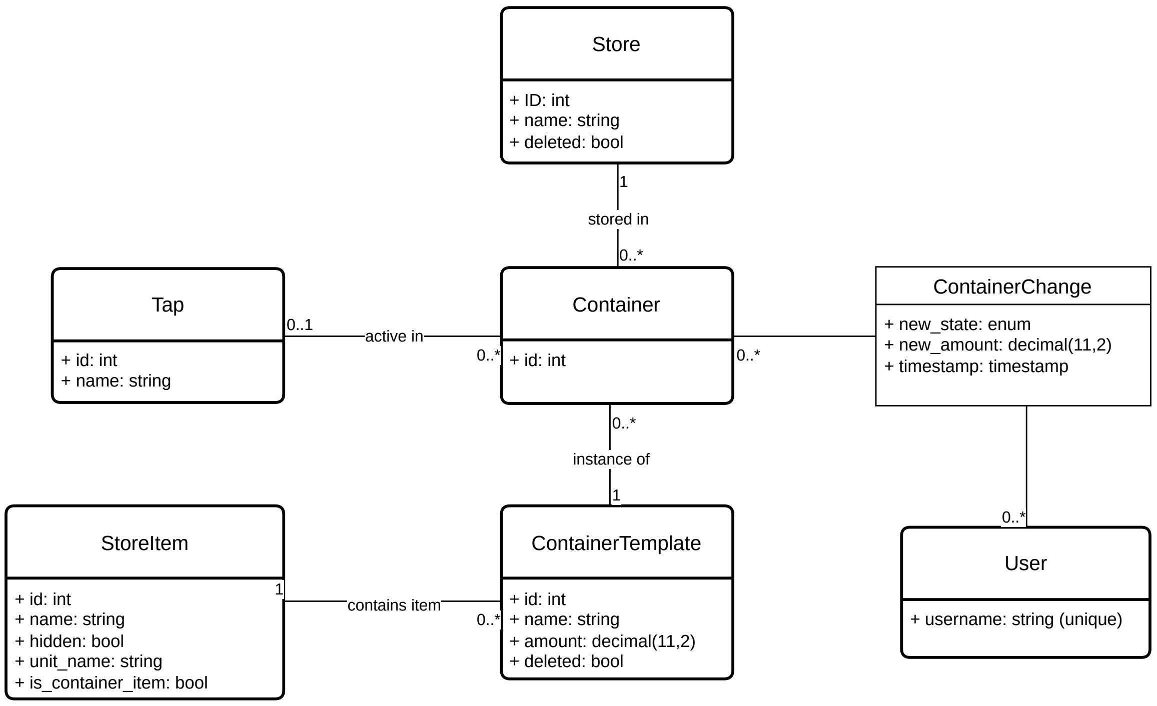


Figure A.4: Entity Relationship diagram for the new KIS Sales – Containers (Kegs for tapped drinks)