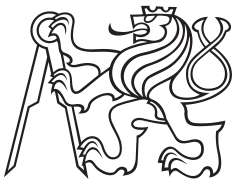
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

Grocery shopping optimization tool

Jan Lindauer

Supervisor: doc. Ing. Stanislav Vítek, Ph.D.
May 2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Lindauer** Jméno: **Jan** Osobní číslo: **486427**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Nástroj pro optimalizaci nákupu potravin

Název diplomové práce anglicky:

Grocery shopping optimization tool

Pokyny pro vypracování:

Navrhněte webovou aplikaci, která na základě uživatelem zadaného nákupního seznamu s potravinami zvolí takový obchod či kombinaci obchodů, aby celková cena nákupu byla nejnižší.
Dílčí problémy, které je potřeba řešit:
1. návrh a formulace úlohy nalezení nejnižší ceny jako ILP problému
2. sběr dat - vysledování komunikace online služeb obchodů a využití této znalosti pro získání dat o cenách a detailech zboží
3. sloučení stejných produktů z různých obchodů pod stejné položky v databázi aplikace
4. návrh a implementace systému - databáze, server, webová aplikace

Seznam doporučené literatury:

[1] CARNELL, John; SÁNCHEZ, Illary Huaylupo. Spring microservices in action. Simon and Schuster, 2021.
[2] CASCIARO, Mario; MAMMINO, Luciano. Node. js Design Patterns: Design and implement production-grade Node. js applications using proven patterns and techniques. Packt Publishing Ltd, 2020.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Stanislav Vítek, Ph.D. katedra radioelektroniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **02.02.2023** Termín odevzdání diplomové práce: **26.05.2023**

Platnost zadání diplomové práce:
do konce letního semestru 2023/2024

doc. Ing. Stanislav Vítek, Ph.D.		prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Datum převzetí zadání

Podpis studenta

Acknowledgements

Chtěl bych poděkovat své rodině a přátelům za podporu během celého studia. Také bych chtěl poděkovat vedoucímu práce za pomoc při tvorbě této práce.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 25. 05. 2023

.....

Abstract

In the face of rapid inflation, individuals are increasingly interested in finding ways to save money on essential groceries without investing significant effort. However, there is currently no comprehensive database of grocery offers available on the market, nor a tool to compare price of entire shopping list across different stores. The goal of this work was to create a comprehensive database of groceries and their prices, along with a web application that would allow the user to optimize a shopping list based on user-defined parameters and compare the total price of a purchase across stores. The problem of finding the lowest price was converted to an integer linear programming (ILP) problem. By leveraging the APIs of several online shopping services and extracting offers from discount flyers using optical character recognition (OCR) technology, a database containing more than 40,000 offers on 25,000 products from seven major supermarket chains operating in the Czech Republic was created.

Keywords: Integer linear programming, Minimum cost flows, Matching problem, Data scraping, Man in the middle attack, Flyer extraction, Optical character recognition

Supervisor: doc. Ing. Stanislav Vítek, Ph.D.
Praha, Technická 1902/2, místnost: B2-719

Abstrakt

Vzhledem k rychlé inflaci lidé stále více řeší, jak ušetřit na základních potravinách. V současné době ale neexistuje žádná ucelená databáze s cenami potravin dostupných na trhu ani nástroj pro porovnání cen celého nákupu v jednotlivých obchodech. Cílem této práce bylo vytvořit komplexní databázi potravin a nabídek, spolu s webovou aplikací, která by uživateli umožnila zoptimalizovat nákupní seznam na základě uživatelem definovaných parametrů a porovnat celkovou cenu nákupu napříč obchody. Problém nalezení nejnižší ceny byl převeden na problém celočíselného lineárního programování (ILP). Využitím API několika online nákupních služeb a extrakcí nabídek ze slevových letáků pomocí technologie optického rozpoznávání znaků (OCR) byla vytvořena databáze obsahující více než 40 000 nabídek na 25 000 produktů ze sedmi největších obchodních řetězců působících v České republice.

Klíčová slova: Celočíselné lineární programování, Úloha toků s minimalizací ceny, Problém párování, Vytěžování dat, Útok Man in the middle, Extrakce dat z letáků, Optické rozpoznávání znaků

Překlad názvu: Nástroj pro optimalizaci nákupu potravin

Contents

Figures

Tables

Chapter 1

Introduction

During the current rapid inflation, many individuals are faced with the increasing prices of essential groceries. They are keen on saving money during shopping without investing a significant amount of effort.

While tools for summarizing and searching currently discounted products exist, there is no comprehensive database that includes products that are not on sale and their prices. Moreover, manually searching for the best offers is time-consuming, making it nearly impossible to find the shop where the total cost of a shopping list would be the lowest. Some people even combine shops to obtain the cheapest combination of offers, which is an even more overwhelming task.

Aiming to change this situation, the purpose of this study is to create a comprehensive database of all products and offers on the market and develop a user-friendly interface to optimize shopping lists. The interface would allow users to compare the total price of their shopping list across different shops and even find the cheapest combination of multiple shops.

A web application has been developed to enable users to manage their shopping lists, which can include specific products from particular manufacturers or generic products without specifying a brand (referred to as product groups in this study). Users select the quantity they wish to purchase for each item, and the application automatically selects a combination of stores to visit and specific products to buy to minimize costs. Users can also specify additional optimization parameters, such as the maximum number of stores they are willing to visit or which supermarket chains to include in the optimization. If users have a membership with a specific chain's bonus club, the application will use membership prices for optimization.

It was necessary to create a database model, populate the database with data, and implement the server and client side of the application. The database model and the client and server side of the application are described in Chapter 2. The problem of finding the lowest purchase price was converted into an Integer Linear Programming problem (ILP). Details regarding the methodology for calculating the lowest price can be found in Section 2.4.4.

Collecting data on product prices and details from each chain and populating the database is a crucial step. The techniques employed for this data collection are thoroughly discussed in Chapter 3. The main data source used

for this purpose is the APIs employed by supermarket chains for their online shopping applications. To understand how these APIs work, the communication of the applications was traced, and this knowledge was applied to obtain product data. The functionality of each API is described in Section 3.1. To keep track of seasonal and discount offers, a product information extractor was designed for marketing discount flyers (Section 3.2). Optical character recognition (OCR) was used to detect text areas and extract product features, while the matching of corresponding product features was formulated as a minimum cost flow problem.

Chapter 2

Application Implementation

This chapter offers a comprehensive overview of the designed system, which consists of frontend and backend.

The backend is detailed in Section 2.1. The Section covers the designed database model, implemented features and interface, and information on deployment. The frontend implementation, which presents a user-friendly interface for utilizing the application's functionalities, is described in Section 2.2.

Section 2.4 describes the optimization process responsible for finding the best combination of shops and offers. Motivation for creation of product groups and their application is discussed in Section 2.3.

2.1 Server and interface it provides

The server provides an interface for managing the user's shopping list, searching for products, and optimizing and managing the user's shoppings. This Section outlines its key features and functionalities.

2.1.1 Server structure

The server is implemented using the Java Spring framework [1]. Its structure is shown in Figure 2.1. It follows the recommended Java design patterns [3] and Spring design patterns [18]. It consists of the following layers:

- Model: Represents the domain entities and defines the structure of the application's data.
- Data Access Objects (DAO): Responsible for handling database interactions, including querying and data persistence.
- Services: Contains the business logic and handles the processing of data between the DAO and REST API controllers.

[1] `https://spring.io/projects/spring-boot`

- REST API Controllers: Manage the communication between the client-side application and the server, providing a well-defined interface for accessing and manipulating the application's data.

The server is connected to a PostgreSQL database [2], where the data are persisted.

Security is maintained using Spring Security, a framework that provides authentication, authorization, and protection against common security vulnerabilities.

To optimize the data transfer between the server and the client, Data Transfer Objects (DTOs) are used. DTOs offer several benefits, such as reducing data transfer, enhancing security by excluding sensitive information, and organizing data for the client.

2.1.2 Database model

The database model of the application is described using an UML diagram shown in Figure 2.2. Key properties of the model are described in the following subsections. All primary keys in the database are automatically generated artificial integer (or long) keys.

Product records

The Product SQL table contains information about two distinct types of products: concrete products and product groups. Products can be recursively grouped under product groups. Product groups are discussed in greater detail in Section 2.3. Concrete product records include package size, name, and optional attributes like brand and barcode. Each product belongs to a specific category, and these categories are organized into a tree structure.

Each product group is associated with a blacklist used when assigning products to groups. The purpose and functionality of group blacklists are also discussed in Section 2.3.

Offer records

The Offer SQL table contains information about offers associated with concrete products. Offer records include the offer's validity, as well as start and end dates, determining the time frame during which the offer is active. Each offer record contains the cost for one package of the associated product and an ID that enables localization of the offer within its source. Two primary types of offers are stored in the table: discount offers and normal offers. Discount offers can additionally include the percentage of the discount applied to the product's original price.

[2] `https://www.postgresql.org/`

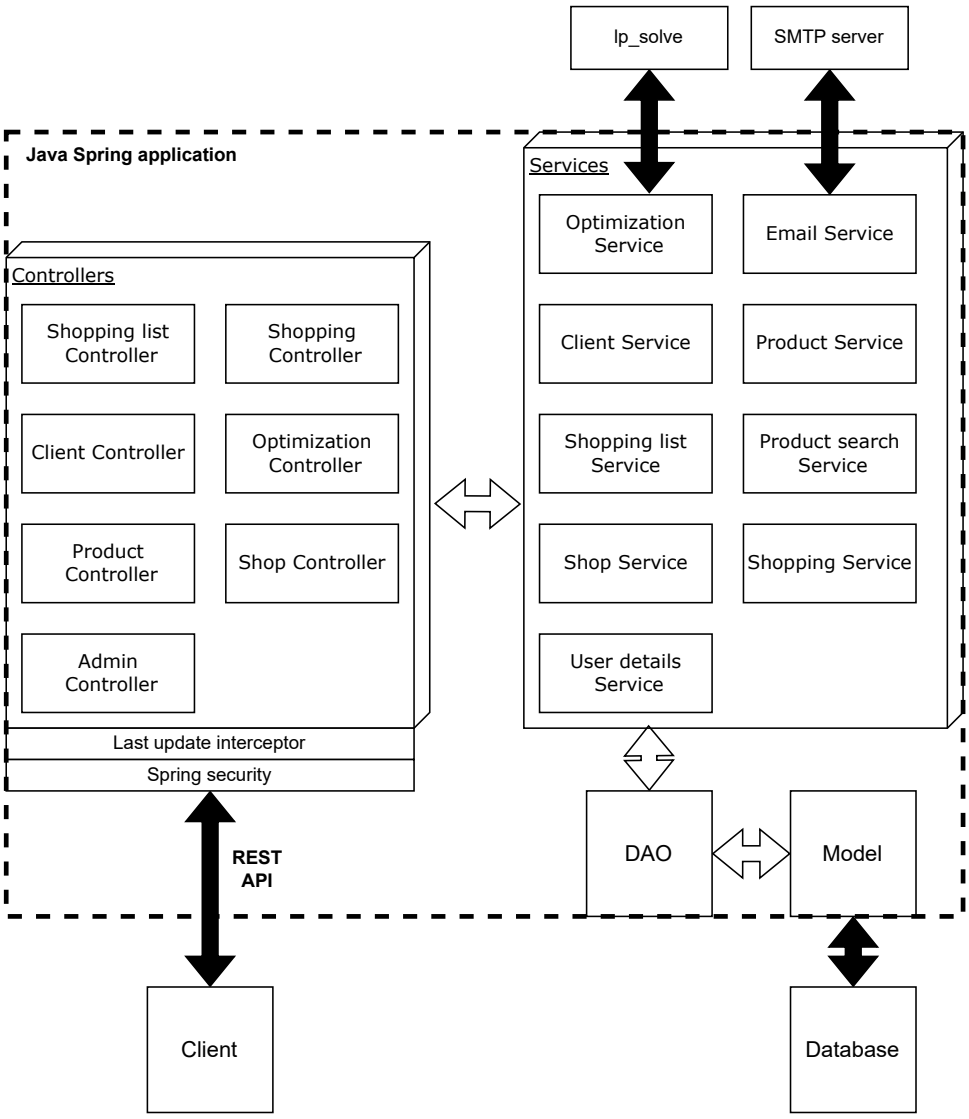


Figure 2.1: Schematic structure of the backend application, the black highlighted arrows indicate communication outside of the main Java Spring application

Shopping List

Each user has a shopping list with any number of shopping list items. A shopping list item can correspond to a concrete product or product group in the database or be a user's plain text note. Each shopping list item can have four states:

- Active: The user needs to buy the item.
- Sale: The user only wants to buy the item if it is on sale.
- Inactive: The user does not currently need the item.
- Hidden: The user had the item in their shopping list but deleted it.

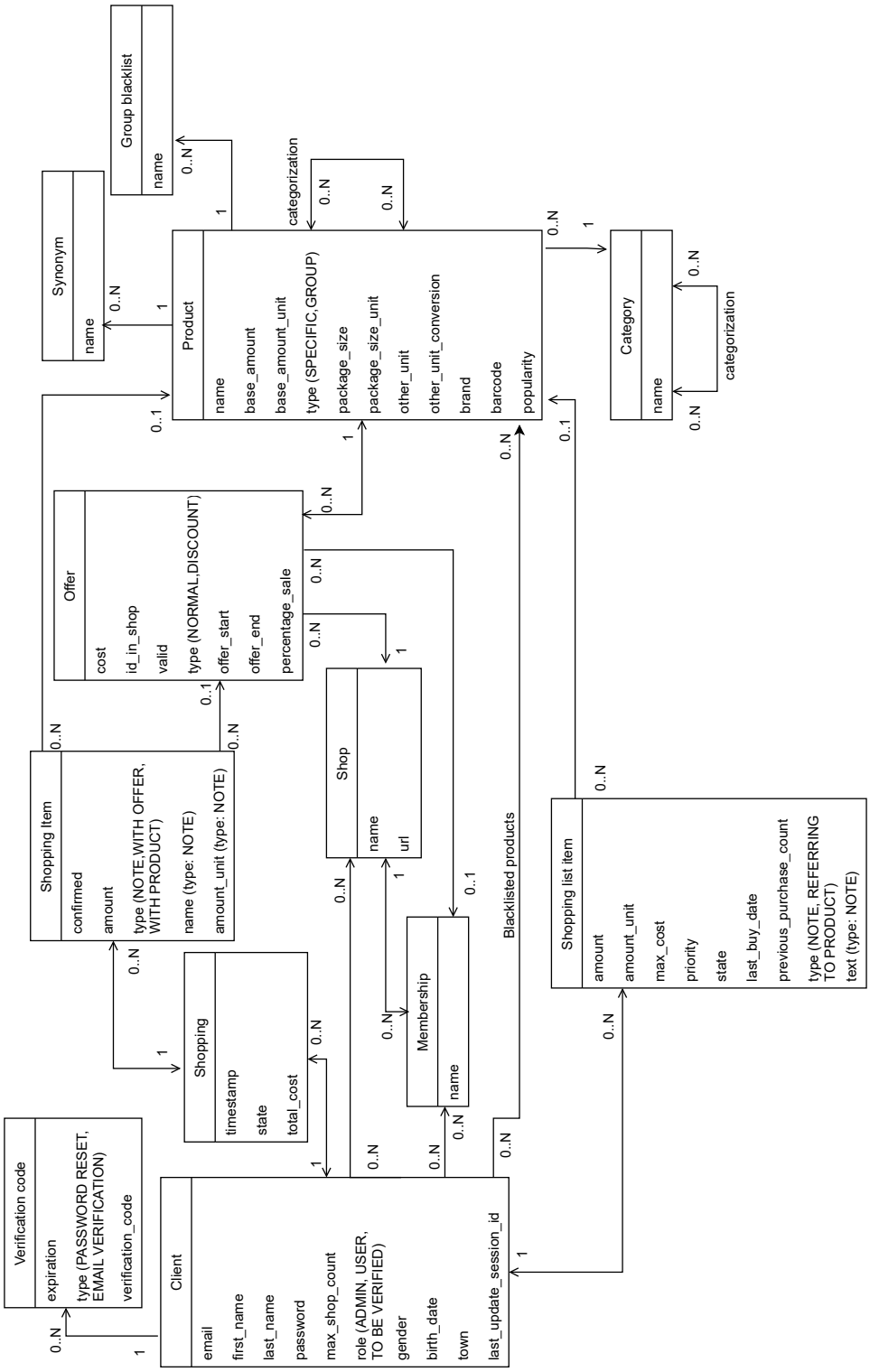


Figure 2.2: UML diagram of the database model.

Users can add items to their shopping list by searching for products by name (2.1.5), writing a text note, or choosing from a list of sale recommendations

(2.1.6). They can change the quantities of items in their list and set maximum costs for products to be included in the optimization process. Additionally, users can allow items to be considered for optimization only if they are on sale, a useful feature for items that the user doesn't need urgently but waits for them to be on sale.

User's shops and memberships

Users can select a list of their preferred shops and benefit membership programs they want to see offers from. These shops and memberships will be used for optimization, allowing users to customize the optimization process to suit their preferences.

Product Blacklist

Users can blacklist specific products, preventing these items from being included in the optimization. This feature accommodates personal preferences and also allows the user to blacklist products that were incorrectly added to a group by the system (Section 2.3).

2.1.3 Optimized Shopping Options

A key feature of the application is the ability to optimize the user's shopping list for the best price. Once the user completes his shopping list, he can trigger the optimization process and receive optimized shopping options. The optimization results include:

- One shopping option optimized for each shop separately.
- One optimal combination of shops that respects the user's restrictions and preferences (if feasible).

For a detailed description of the optimization process, see Section 2.4.

Users can then start shopping by selecting one of the provided shopping options. After that, a new "shopping" record is created in the database with shopping items. There are the following types of shopping items:

- Note - active notes from the shopping are added as text notes without reference to the product nor offer.
- Optimized item - created for each optimized shopping list item, refers to the concrete offer found by the optimization.
- Missing item - items for which the optimization didn't find an appropriate offer satisfying the optimization constraints, they refer to the product instead.

As users shop, they can mark ongoing shopping items as bought or unmark them if needed. Once the shopping is completed, they can finalize the shopping.

Users can view their previous shoppings, providing a historical record of their purchases.

2.1.4 User Management

The web application supports user registration, including email verification, login, password update or reset, and logout. This ensures that users can securely manage their accounts and personal information.

When the user registers, a unique code is generated and combined with the user's email into a verification URL, which is sent to his email using the public SMTP server. This link is valid for a limited time period. After this period, the user has to request the link to be resent. Until the user is verified, his role is set to "Not verified", and he cannot use the full functionality of the application. After verification, his role changes to "User", and he can start using the application.

Similarly, when the user requests to reset his password, a password reset link is sent to his email.

The "Verification code" SQL table is used to store the unique email verification codes and password reset codes, as shown in the UML diagram.

2.1.5 Product Search

Users can search for products by name. The search query is split into a word set, and the search results include products with names that contain all words (or words starting with search words) present in the search query word set.

The search results are sorted to provide the most relevant products. The factors used for sorting are (listed by importance):

1. Product group vs. specific product: Product groups (except for empty groups) are considered more relevant than specific products, as users usually don't need a specific brand and packaging.
2. Shop count: Products available in a larger number of shops are considered more relevant.
3. Alphabetical sorting.

The five most relevant search results are presented to the user.

To ensure fast search results, the server caches a search result for each product in the database on server startup. At startup, it also sorts all the search results, so it doesn't have to be done on each search request. This process might take some time during initialization, but caching is crucial to ensure acceptable search time. Additionally, the system can save search items in a file to speed up the startup process. Whenever the product database is updated, the cache needs to be evicted and recreated.

2.1.6 Sale Recommendations

Users often find it useful to see what is currently on sale when creating their shopping list, deciding between shops, or during shopping. Therefore, the app maintains sale recommendations, and the frontend displays them when convenient (see Section 2.2.2).

Due to a large number of sales, sale recommendations only contain discounted product groups (more on groups in Section 2.3). Moreover, the discounted offer has to be the cheapest among the group in order to be displayed within the recommendations.

To emphasize the most interesting sales, the recommendations are sorted. A rank is computed for each recommendation based on the following factors, and the recommendations are then ordered accordingly:

1. Category importance: Products belonging to more popular categories are ranked higher in the search results, as they are generally of higher interest to users.
2. Percentage sale: Products with a higher sale percentage are considered more relevant, as they offer better deals for users.
3. Shop count: Products available in a larger number of shops are considered more relevant.
4. Subproducts count: A greater number of subproducts of a product group increases its relevance.

Similar to the product search results, the sale recommendations are cached to ensure a fast response.

2.1.7 REST API

The server exposes REST API to the Internet which is used for communication between frontend and backend. It is documented in detail in Attachment 5.1. Most of the API endpoints are available only to signed-in users.

There are also a few endpoints that are only available to an administrator user. This is the case for the endpoint that can be used to refresh all caches of the Java Spring Application after changes were made in the database from outside of the application. This endpoint is called after any product database updater described in Section 3 finishes.

2.1.8 Deployment, Runtime environment

The application is divided into three Docker containers [3]. All three containers are based on Debian [4] images. The first container runs Tomcat web server

[3] `https://www.docker.com/`
[4] `https://www.debian.org/`

[5] hosting the main Java Spring application. The second container is the application PostgreSQL database.

In the third container are all scripts responsible for data mining. There is one scraper for each supported shop written in Node.js [6] and one separate scraper for data extraction from flyers in Python. These data acquisition scripts are described in Chapter 3 in detail.

The containers are orchestrated using docker compose[7]. The entire application is hosted in the cloud. The application supports HTTPS and only port 443 (and port 80 in order to redirect to 443) is exposed to the Internet.

2.2 Frontend

A web application was developed to allow users to interact with the interface provided by the server. It was developed using React [8] and designed with a responsive layout using Tailwind CSS [9] to provide an optimal user experience across a wide range of devices, from desktops to mobile phones.

2.2.1 Server Request Management Layer

One of the essential components of the frontend is a layer for handling server requests. This layer ensures that all requests to the server are queued and executed in the order they were initiated, preventing potential conflicts or inconsistencies in the application state.

The server request management layer provides the following functionalities:

- Automatic retries after network failures: If a request fails due to network issues, the layer automatically retries the request after a specified delay, improving the application's resilience to connectivity problems.
- Ensuring consistency: When the user tries to make any state-modifying request and the current device is not the one that made the last edit, the network layer handles a message from the server informing that this situation happened. It fetches all user data from the server and then repeats the original request. This functionality ensures that users have access to the most up-to-date information and helps to prevent conflicts and data inconsistencies that may arise when multiple devices are used to access and edit the same data.
- Logout on authorization failure: If a request fails due to an authentication or authorization issue, the layer automatically logs the user out and redirects him to the login page.

[5] https://tomcat.apache.org/
[6] https://nodejs.org/en/
[7] https://docs.docker.com/compose/
[8] https://react.dev/
[9] https://tailwindcss.com/

- Visualizing loading: If the request queue is not empty, it is indicated to the user with a loading animation.

2.2.2 Main Pages

The application consists of several main pages, each serving a specific purpose and providing different functionalities to the user.

Shopping List Page

The shopping list page allows users to manage their shopping list, including adding, removing, or inactivating items, changing quantities, setting the maximum cost of the product to be included in the optimization, or allowing the item in the optimization only if it's on sale. There are three subpages:

- Active shopping list items
- Inactive shopping list items – items previously present in the shopping list, but not marked as needed at the time
- Sale recommendations – more on that was discussed in Section 2.1.6

The offer with the best value per amount is displayed alongside each shopping list item to assist the user in deciding whether he wants the item. Screenshots of the shopping list page are shown in Figure 2.3.

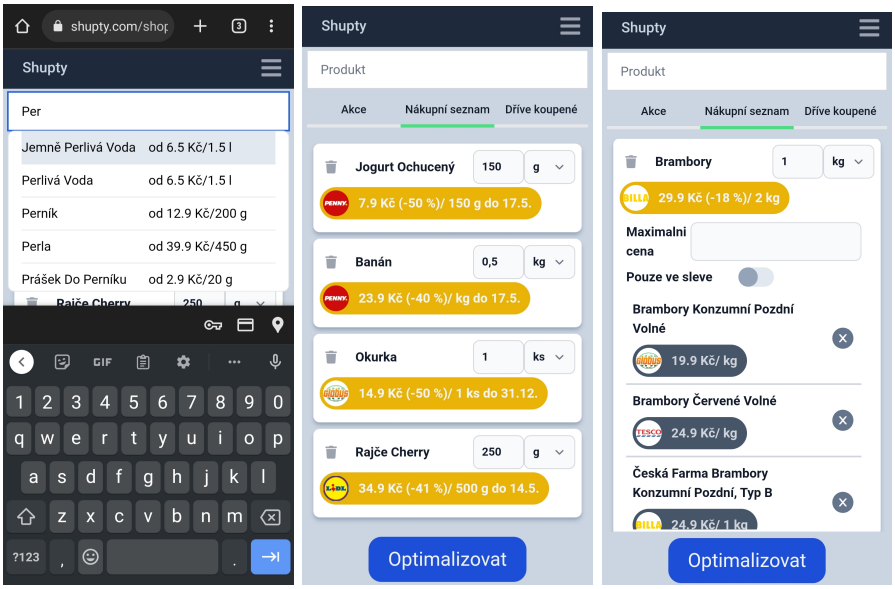


Figure 2.3: Shopping list page. From left to right:
a. autocompletion when searching for a product to add,
b. shopping list items,
c. details of a shopping list item (additional constraints and associated subproducts)

Optimization Parameters Page

On this page (shown in Figure 2.4), users can configure the optimization settings, such as selecting the shops to be used for optimization, setting the maximum number of shops to visit, and choosing the loyalty programs to consider for offers. After setting the parameters, the user can trigger the optimization.

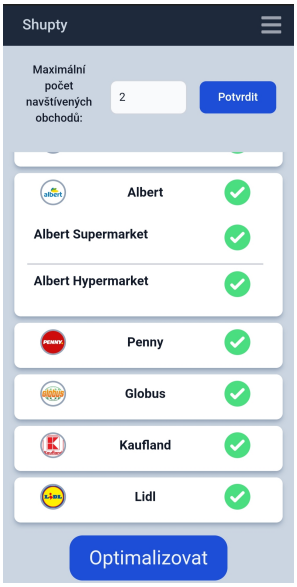


Figure 2.4: Parameters page. Choosing shops, memberships and maximum number of shops

Optimized Shopping Options Page

On this page (shown in Figure 2.5), optimized shop combinations including concrete products to buy are shown. There is one shopping option per shop and one shopping option that combines up to N shops, where N is the maximum number of shops to visit set by the user in the previous step. Under each shopping option are displayed recommendations for products that are on sale in the corresponding shops (Sale recommendations – Section 2.1.6). Users can additionally add any of these products to their shopping. After selecting one of the shopping options, this shopping becomes an ongoing shopping and the user is redirected to the "Current shopping page".

Current Shopping Page

On this page (shown in Figure 2.6), users can view and interact with their ongoing shopping, marking items as bought and finalizing the shopping once completed. Again, a list of sale recommendations is displayed below shopping items.

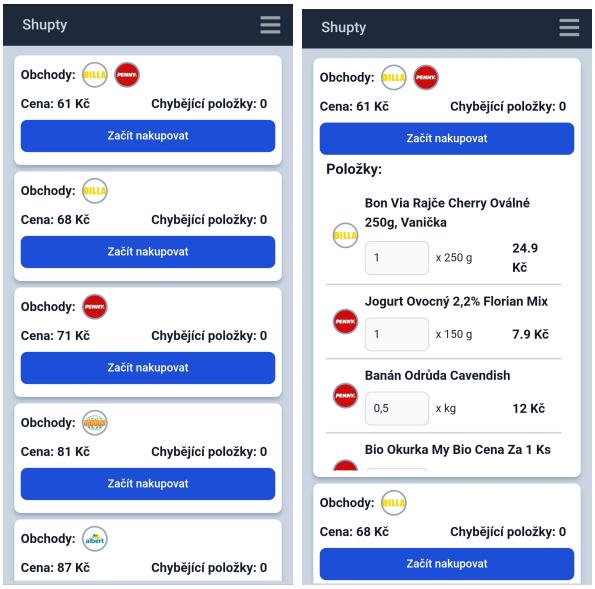


Figure 2.5: Shopping options page. One option combining more shops and one per each shop, options are expandable (right figure)

Figure 2.6: Current shopping page. User can (un)mark items as bought, shopping suggestions are shown below the shopping list

Shopping History Page

This page displays a history of the user's previous shoppings, including the items purchased, the total cost, and the date of each shopping.

2.3 Product Groups

In many cases, users want to purchase groceries without specifying an exact product or brand. To accommodate this, product groups are introduced, allowing users to choose a group of similar products instead of specifying a specific item. This section discusses the creation and utilization of product groups in the application.

2.3.1 Defining Product Groups

A product group is a collection of similar products. For example, the "Mozzarella" group would contain all the mozzarella products available on the market.

A list of product group names is created manually, representing items that users might want to purchase. Along with the group name, information about the group categorization, group name synonyms, and a blacklist of words that cannot be present in the product name are also provided. The blacklist ensures that only relevant products are included in the group.

2.3.2 Assigning Products to Groups

When a product is added to the database, it is automatically assigned to relevant product groups. The following logic is used to determine which product groups the product should be added to:

1. A product cannot be added to a group with a different categorization.
2. If the product name contains any blacklisted words from a group, it does not belong to that group. For example, let's consider the product group "müsli", which has a blacklist containing the word "stick". When users include "müsli" in their shopping list, they aren't interested in buying "müsli stick", so the product won't be added to the "müsli" group.
3. Otherwise, if the product name contains all the words of a group name (or any synonym of the group name), it is added to the group.

2.3.3 Utilizing Product Groups in Optimization

When a user adds a product group to their shopping list and requests optimization, the system will search for the best-priced product within the selected group.

The optimizer then incorporates the selected product into the optimized shopping, ensuring that the user gets the best value while still meeting their grocery needs. The product group functionality allows for greater flexibility and personalization of the shopping experience, as users can choose from a variety of products without being limited to specific brands or items.

2.3.4 Challenges and Improvements

The implementation of product groups and matching presents several challenges. Manually creating the list of product group names and managing their categorization, synonyms, and blacklist is time-consuming and may not cover all possible products. Additionally, the product matching logic may not always accurately assign products to their respective groups, leading to potential discrepancies in the optimization results.

To address these challenges, machine learning techniques and natural language processing could be employed to automatically identify and categorize similar products and create product groups. These techniques could help to improve the accuracy of product matching and simplify the process of creating and maintaining product groups.

2.4 Optimization of Shopping Costs

This section discusses an optimization approach to minimize shopping costs by selecting the best offer for each item on a shopping list across different shops while considering user-defined parameters.

The user-defined parameters are the following:

- List of shops considered in the optimization
- List of user's memberships (offers requiring other memberships are omitted from optimization)
- Maximum number of shops visited (N)
- Maximum price for an item
- Option to only allow discount offers for an item

The first step in the optimization process involves finding the best offer for each item on the shopping list in each selected shop separately. The process of finding the best offer is described in Section 2.4.1.

Additionally, if a shopping list item refers to a group of products, it is necessary to recursively search for all specific products belonging to this group and choose the cheapest one (which might not be the same one across all user-selected shops).

Section 2.4.3 explains how shopping list items with user constraints for maximum price or discount-only offers are handled.

After these initial steps, optimization for selecting the cheapest shop combination can be triggered. This optimization is described in Section 2.4.4.

2.4.1 Finding the best offer for a specific product

Each specific product is stored in the database with information about its package size. Note that not every product is packaged; some products are offered by pieces or by weight. However, to simplify terminology, let's also

use the term package size for piece size and for 1 kilogram of a weighted product. All offers linked to a specific product state the cost for one package.

The user can input the desired amount in various units, such as liters, milliliters, kilograms, grams, pieces, or a number of packages. To find the cheapest offer, it is crucial to convert the user-input amount to the units in which the package size is stored. Then, the number of packages needed to satisfy the user's request can be calculated (for weighted products, this value doesn't have to be an integer). The computation is described in Section 2.4.2. By multiplying the number of packages by the price per package, the total cost of the item is obtained.

2.4.2 Computing required number of packages

The decision tree for computing the number of packages needed is depicted in Figure 2.7. It might be necessary to convert between units that are unrelated, like pieces and weight. To allow this conversion, another unit O can be stored in the database along with conversion coefficient t_O, where t_O is the number by which if we multiply the amount in unit O, we get the amount in the package size unit.

For example, let's assume that the user wants to buy 10 apples and that the apples have a package size of 1 kilogram. Then the database would store that there is another unit, pieces, with t_O equal to the weight of one apple in kilograms. By multiplying 10 apples by the apple weight, we would get how many kilograms are needed to satisfy the user's request.

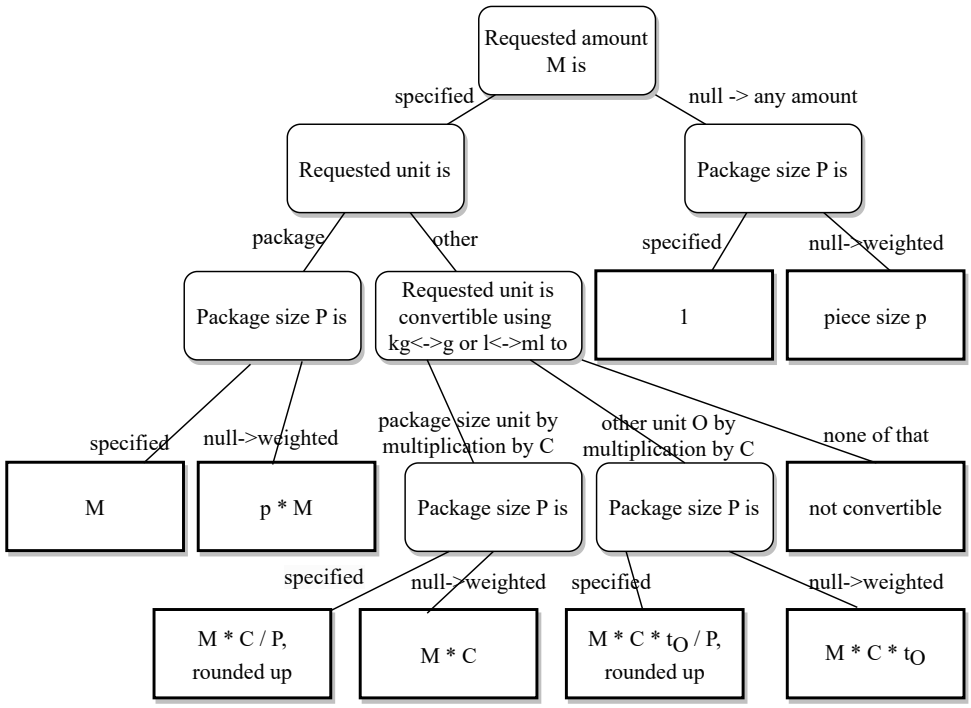
The t_O can be stored together with each specific product, but if not provided, the t_O of a product group the product belongs to is used instead.

2.4.3 Optional shopping list items

We need to address shopping list items that the user marked to be included only if they are on sale or cheaper than a user-set maximum cost. If all shops used in the optimization provide offers satisfying these constraints, the item is included in the ILP optimization. Otherwise, it is omitted from it and added additionally after the ILP optimization is completed. Of course, the item is only added to shopping options that offer the item on sale or at a low enough price, as the user requested.

2.4.4 Formulation of the ILP Problem

The problem of finding the lowest price for the total purchase in at most N stores can be formulated as an Integer Linear Programming (ILP) problem [20]. Let's arrange the selected specific products with the best offers and their prices from the previous optimization steps into a matrix A, with P (number of products in the shopping list) rows and S (number of shops included in the optimization) columns. $a_{p,s}$ represents the price of the p-th product in



t_O conversion coefficient for conversion to other unit O
M user requested amount provided in requested unit
P package size
p piece size (if other unit O pieces is available, piece size is t_O otherwise it is guessed)

Figure 2.7: Decision tree for calculating the product amount the user should purchase

the s-th store. Then the optimization problem can be formulated as follows:

$$\begin{aligned} \min & \sum_{p\in 1\ldots P} \sum_{s\in 1\ldots S} x_{p,s} a_{p,s} \\ \forall p & \sum_{s\in 1\ldots S} x_{p,s} = 1 \\ \forall s & \sum_{p\in 1\ldots P} x_{p,s} \leq z_s P \\ & \sum_{s\in 1\ldots S} z_s \leq N \\ \text{variables:} \quad & x_{p\in 1\ldots P, s\in 1\ldots S} \in \{0,1\}, \quad z_{s\in 1\ldots S} \in \{0,1\} \\ \text{parameters:} \quad & a_{p\in 1\ldots P, s\in 1\ldots S} \in \mathbb{R}^+, \end{aligned} \tag{2.1}$$

where $x_{p,s}$ determines whether item p in store s is selected for purchase, and z_s determines whether the user should visit the s-th store. The first constraint ensures that each product is purchased in exactly one store. The second constraint indicates that if at least one product is purchased in the s-th store, the user must visit that store. The last constraint specifies that the number of stores the user visits is at most N.

By solving this optimization task, we will obtain an optimal solution indicating which shops to visit (the ones with $z_s = 1$) and where to buy each of the products (the product p should be bought in shop s if $x_{p,s} = 1$).

If the optimization fails to find a solution, only shopping options optimized for each shop individually by the previous optimization steps are presented to the user.

2.4.5 Implementation

To solve the ILP optimization task, the lp_solve [13] library is used. The Java ILP wrapper [10] is employed to trigger the optimization from the main Java application.

[10] `https://javailp.sourceforge.net/`

Chapter 3

Data acquisition

To effectively optimize shopping and deliver maximum value to users, a comprehensive database of products is essential.

This chapter outlines the data sources and extraction methods used to gather product information. Data is acquired from online services provided by shops and from discount flyers.

Section 3.1 details the process of obtaining data from online services, including reverse engineering the underlying APIs and utilizing it for data scraping. The design and orchestration of data scrapers is also explained.

Section 3.2 describes how data is extracted from discount flyers, covering text area extraction, product feature detection (such as name, price, and quantity), and the methods used for matching corresponding features.

As the same products may be offered in different shops under various names, matching offers for identical products is crucial. Section 3.3 elaborates on the methods used for this purpose.

3.1 Data Scraping

Various stores offer different online services, such as online delivery e-shops, shopping list tools, or self-service shopping tools. These services provide users with information about available products, along with related details, including prices. The underlying APIs employed by these services were successfully utilized to access this information.

In particular, data are acquired from the following services:

- Albert domů zdarma [1] (online shopping tool)
- Tesco online [2] (online shopping tool)
- Můj Globus [3] (self-service shopping application, shopping list tool)
- Billa online [4] (online shopping tool)

[1] `https://www.albertdomuzdarma.cz/`
[2] `https://nakup.itesco.cz/groceries/`
[3] `https://www.globus.cz/muj-globus.html`
[4] `https://shop.billa.cz/`

- Kaufland application [5] (self-service shopping application)
- Penny domů [6] (online shopping tool)
- website listing discounts in Kaufland [7]

The first step involved reverse engineering the underlying APIs. To trace all web application communication in HTTPS unencrypted form, browser developer tools were used. The method utilized for reverse engineering mobile applications is described in Section 3.1.1.

After analyzing these APIs, a specialized scraper for each of them was developed. These scrapers were implemented using Node.js. The functionality of each scraper is explained in subsections 3.1.2 - 3.1.8.

3.1.1 Mobile Apps APIs reverse engineering

To trace the communication of a mobile app, a Man in the Middle MITM attack can be used. In this work, Mitmproxy tool [17] (3.1.1) running on a PC was used to intercept mobile phone traffic.

Mitmproxy [17]

Mitmproxy [17] is a tool used to intercept and manipulate network traffic in a MITM attack scenario. It acts as a proxy and a Certificate Authority (CA) by generating SSL/TLS [19] certificates on the fly to sign traffic that it intercepts. Mitmproxy generates new SSL/TLS certificates for the destination server, signs them with its own CA certificate, and intercepts and decrypts the traffic passing between the monitored device and the server. To use mitmproxy, its CA certificate must be installed on the user's device as a trusted CA.

3.1.2 Albert products scraping

To acquire the available products and their prices from the Albert chain store, the API used by their web shopping application, "Albert domů zdarma", serves as the data source. The common elements of this API can be found in Attachment 5.2.1.

The offered products are organized into categories. Using the browser's developer tools, it was traced that the web application uses a request 5.2.2 to obtain all available product categories. These categories are further divided into several levels. By navigating through all the categories at a certain level, it is possible to access all the products in the system.

Request 5.2.3 is used to retrieve a product page associated with a specific category. The response to this request contains basic product information,

[5] `https://prodejny.kaufland.cz/nabidka/aktualni-tyden.html`
[6] `https://www.pennydomu.cz/`
[7] `https://www.kaufland.cz/aktualne/servis/mobilni-aplikace.html`

including price and identifier. Additional product details can be obtained by requesting a specific product using its identifier through request 5.2.4.

The scraper developed in this study locates all categories at the lowest level of the categorization hierarchy and identifies all products within each category by traversing the product pages of each category. Given that scraping by page is significantly faster than by product details, the product information are retrieved by page (request 5.2.3). The supplementary information available in the product details is not required for adding the product to the database.

3.1.3 Tesco Products Scraping

The Tesco online service primarily generates web pages statically on the server side, with product lists and product details sent to the client in HTML format. After acquiring the first static HTML page, it switches to API calls which allows the web application to acquire category tree, product lists by categories or product details in JSON format.

Although it is possible to scrape data through this API, the API security is challenging to understand. Consequently, a more straightforward approach of scraping the statically generated pages and consequent HTML parsing is used to extract the data. The category tree, however, was obtained in JSON format by intercepting the web application communication.

The designed scraper shares a similar approach to the Albert scraper, navigating through all pages of all categories at the lowest category level to collect the necessary information. The primary difference between the two scrapers lies in the data format used for each platform. Despite this distinction, both scrapers are similar.

3.1.4 Billa Products Scraping

The Billa Online web application stores information about available categories as constants in JavaScript. They were manually extracted from the code. To retrieve products in any category, request 5.5.1 is used. The scraper again sequentially traverses through all the most specific categories to obtain all available products, along with their respective prices.

3.1.5 Penny Products Scraping

The Penny domů service only provides product information in HTML format. The HTML pages include a navigation bar, containing URLs that lead to all the available product categories. When a specific category URL is accessed, a paginated list of all products belonging to that category is displayed. These product lists are generated on server-side and delivered in HTML format.

The implemented scraper initially parses list of available categories and their corresponding URLs. Subsequently it navigates through each category, page by page, extracting product details directly from the HTML.

3.1.6 Kaufland products scraping

Kaufland offers an app for self-service shopping, which allows users to scan barcodes during their shopping journey and pay at the end. This app provides product details and prices upon scanning each barcode. The goal was to make use of the underlying API utilized by the app to collect the relevant product information and prices.

Bypassing Certificate Pinning

The Kaufland application is secured against MITM attacks. This was discovered by first attempting to intercept the HTTPS traffic just by installing the mitmproxy CA certificate on the device. This resulted in network failures of all the requests the application made.

By further analysis, it was discovered that the application contains a list of pinned trusted certificate authorities directly in the application package (for Android, this is in APK format [8]). To bypass the pinning, the Android version of the package (APK) was modified by using the following steps:

1. First, the APK was acquired using the Android remote debugging tool ADB [9].
2. The APK was unpacked using Apktool [10].
3. The content of the APK file was examined and pinned certificate files were localized.
4. The CA certificate of the mitmproxy [17] running on the PC was acquired.
5. The mitmproxy's CA certificate file was converted to the same (BKS [11]) format as the pinned certificates [12].
6. The certificate files in the unpacked application were replaced with the ones from the previous step.
7. The modified application was packed to APK using Apktool.
8. The resulting APK was signed using Uber APK Signer [13] to allow its installation on Android devices.

After performing these steps, the modified application's traffic could be intercepted. Moreover, the modified application only trusts the mitmproxy's CA certificate with its HTTPS communication.

[8] https://developer.android.com/guide/components/fundamentals
[9] https://developer.android.com/tools/adb
[10] https://ibotpeaches.github.io/Apktool/
[11] https://www.bouncycastle.org/
[12] https://gist.github.com/wbroek/cd87d161b52d0ddba08d
[13] https://github.com/patrickfav/uber-apk-signer

Reverse Engineering the API

To gain access to API functionalities, it was first necessary to log in through the app. The app uses OAuth [5] for authentication. By monitoring the communication between the app and server using a man-in-the-middle (MITM) approach, several key values necessary for taking over the session were obtained: customer ID, store ID, cidaas ID (an implementation-specific ID referring to the Single Sign-On session), client ID, and refresh token. These values are required for making requests to the API.

Using the refresh token and client ID, an access token can be obtained using request 5.6.1. With the access token, customer ID, and cidaas ID, a request 5.6.3 can be made to initiate a new shopping session, acquiring the basket ID and BusinessServerSession ID (implementation-specific).

Once the basket ID and BusinessServerSession ID are available, requests can be made to add an item with a specific barcode to the basket (request 5.6.5). The server responds with one of the following:

1. Product details, including the price of a product with the corresponding barcode, if it exists.
2. Packaging options if multiple options are available. After selecting one of the packagings and repeating the original request, product details are returned.
3. A "not found" code if the product with the barcode is not found.

Items can be removed from the cart in a similar manner as they were added using request 5.6.4.

Scraping Strategy

To simplify the scraping process and avoid replicating the entire OAuth login process consisting of multiple requests, the latest refresh token observed during the communication was used to take over the session for the scraper.

The scraping script behaves like a client using the app to check prices. It performs the following steps:

1. It uses the last observed refresh token to obtain an access token and a new refresh token and saves both tokens.
2. Initiates a shopping session.
3. Sends an "add to shopping cart" request 5.6.5 with a barcode to obtain product details.
4. Waits for a few seconds, simulating a user interaction delay.
5. Removes the item from the basket using request 5.6.4.
6. Waits for a few seconds, simulating a user moving to another item.

7. Repeats steps 3-6 for different barcodes, up to 50 times.

This scraping process is executed three times every non-holiday day to gather product information. This strategy allows efficient product data scraping while minimizing the risk of detection or suspicion.

Barcodes obtained from the Globus scraper (Section 3.1.8) were used as the scanned barcodes (the barcodes referring to products with the Globus brand and their other sub-brands being omitted). Consequently, only products that are also available in Globus were obtained this way. Kaufland-specific brands, as well as products with store-specific barcodes such as bread, fruits, and vegetables, are not present in the database.

It is possible to manually collect a list of barcodes for the missing Kaufland-branded products. However, store-specific barcodes often change due to the seasonal nature of fruits and vegetables and the varying availability of such products. Moreover, the barcode scanning API is not suitable to keep track of weekly discounts as it might raise flags if too many products are scanned at a time or too many shoppings are made daily.

The most effective way to keep track of seasonal offers is by monitoring discount flyers. Kaufland provides their flyers in HTML format. The scraping of these flyers is described in Section 3.1.7.

3.1.7 Kaufland Sales Scraping

Kaufland provides information about their ongoing sales and special offers on their website. To extract sales data, a simple scraper was designed to download the discount pages in HTML format. The product names, prices, categorization, and offer start and end dates are extracted by parsing the HTML content.

This scraper is useful as it keeps track of the discounts and seasonal offers, which cannot be maintained by the barcode scanning API (Section 3.1.6).

3.1.8 Globus Products Scraping

Similar to Kaufland (Section 3.1.6), Globus offers an application called Můj Globus that enables users to utilize their phones as self-service barcode scanners. This app also features a shopping list function, which allows users to add specific products using a search API. As with the Kaufland application, mitmproxy was employed to investigate the application's API. In this case, the application did not have pinned certificates, enabling direct MITM interception of the original application.

To use the APIs, an access token must first be obtained by logging in. The application employs request 5.3.2 to log in, which returns the access token and a refresh token. The refresh token can later be used to renew the access token using request 5.3.3.

Shopping List API

With the access token, request 5.3.4 can be performed to search for a product by name. The matched product names are returned, including product identifiers (referred to as VANRs). By utilizing these identifiers in request 5.3.5, the API provides detailed product descriptions for the corresponding products, including price, name, categorization, barcode, amount, and other details (see Attachment 5.3.5 for the detailed response format).

Scraping Strategy. The API provides access to all the necessary information. The remaining challenge is to create a set of search phrases to find as many products as possible. Fortunately, this is not an issue, as product names from the online delivery services of other shops and a manually created set of product groups are already available. By using these names as search phrases, over 12,000 products were found through the Globus API. Since this is a search API, sending a series of requests in a row should not raise suspicion. Thus, the scraper makes all the search requests in a row with short delays between them. The entire database refresh takes less than an hour.

Self-Scanner API

The Shopping List API was released later than Self-Scanner API, so the scraper was initially implemented using this API. It operates similarly to the Kaufland scanner API (Section 3.1.6). First, a shopping cart must be created using request 5.4.2. This request returns the token required to obtain product information. Then, request 5.4.4 with any barcode can be sent to the server, which will respond with information about the corresponding product and its price. To complete the shopping, request 5.4.3 is used.
This API can be used to obtain product information, but as it is more likely to be monitored and requires a set of barcodes to scan, the Shopping List API (3.1.8) was used instead when it was released.[14]

3.1.9 Scraper scheduling

All scrapers are orchestrated using cron scheduler [15]. The Tesco, Globus, Billa, Penny, and Albert scrapers run weekly in the night or morning hours. The flyer scraper (will be discussed in Section 3.2) runs a few times a week, but only if any new flyer is found. Since the Kaufland scraper is based on the scanner application API, it has to behave like a person going through the shop to avoid attention. So it is only scraping a small set of products in each run and runs multiple times a day.

[14] Globus also released its online e-shop, but at the time of the release, the Shopping List API scraper had already been created.
[15] `https://linux.die.net/man/5/crontab`

3.2 Data extraction from flyers

While a majority of prices can be obtained through scraping of online shop services, some sales are exclusive to physical stores. The most convenient method to look up all sales is by examining discount flyers. To avoid doing this manually, extraction scripts were needed.

Up-to-date flyers from large supermarket chains are available online. However, they are often presented as unstructured PDF files, with only a few shops (Penny, Billa) offering structured PDF versions.

Research has already been conducted with the aim of extracting data from promotional flyers [10, 4, 8]. Most studies utilize OCR followed by post-processing [10, 8]. This study employed OCR in combination with custom pre-processing, post-processing, and robust product name matching with price and quantity.

3.2.1 Preprocessing

Initially, the PDF files are split into individual images by page, as the pages are unrelated with each other (except for offer start and end dates) and all the information about each product is on a single page.

For Lidl shop [16] flyers, OCR initially had difficulty extracting prices. Prices are displayed in a bold white font on a red background and OCR struggled to recognize it as a text. To mitigate this issue the input is further preprocessed. The red color areas used in the flyer as a price background are masked, resulting in a black-and-white image. An example of a preprocessed image is shown in Figure 3.1. Since OCR is better trained to recognize black text on a white background [16], this adjustment led to improved results (Figure 3.2). Additionally, this step preserved price-related text while removing most non-price texts. Consequently, the identification of price-related text became more straightforward, leading to enhanced price detection.

3.2.2 OCR

Initially, the open-source Tesseract [15] was considered for Optical Character Recognition (OCR). However, the results were unsatisfactory (Figure 3.3), as Tesseract [15] is primarily designed for extracting text from formatted documents with single block of text (like books). The challenge with flyers is the presence of multiple text areas scattered throughout the page, having various fonts and font sizes. Exhaustive training of Tesseract [15] would be required for each flyer format to improve the detection, which would cost hours of annotating.

Consequently, the commercial Google Cloud Vision API [12] was tested. The produced results were significantly better (Figure 3.3). Thus, this API was chosen for further use in this study.

[16] `https://www.lidl.cz/`

Figure 3.1: OCR Preprocessing - on the left is the original image, on the right is the image after masking just red color (flyer source: `www.lidl.cz`)

Figure 3.2: Effect of OCR Preprocessing - it is clearer for the OCR that the number 7 is a text after preprocessing (right figure), green bounding boxes show detected text areas (flyer source: `www.lidl.cz`)

3.2.3 Searching for ROIs (Regions of Interest)

Google Cloud Vision API OCR [12] outputs data in the format shown in listing 3.1.

Figure 3.3: Comparison of text areas detected by Tesseract OCR [15] (left) the ones detected with Google Cloud Vision API OCR [12] (right) (flyer source: `www.lidl.cz`)

Listing 3.1: Google Cloud Vision API OCR output format

```
{"responses": [
    {"fullTextAnnotation":{
        "pages": [{
            "blocks": [{
                "boundingBox": {
                    "vertices": [{"x":x, "y":y},...]
                },
                "paragraphs": [{
                    "boundingBox": {...},
                    "words": [{
                        "boundingBox": {...},
                        "symbols": [
                            "boundingBox": {...},
                            "text": "W",
                        ]
                    }, ...],
                }, ...]
            }, ...]
        }, ...]
    },...}
]}
```

The output includes each detected character with its bounding box, organized hierarchically into words, paragraphs, and blocks, which also have their respective bounding boxes.

The main goal is to extract the product name, discount price, and amount. While extracting non-discount prices would be useful, their detection is less accurate as they are crossed, which could result in numerous errors. Therefore, the focus remains on obtaining the mentioned three pieces of information.

The exact format varies depending on the store, but most flyers share a similar pattern. Prices are displayed in a large font (its size slightly differs, but is in store-specific range) with specific background colors. To identify price words (using Google API terminology), a cascade of filters was developed, which words must pass through to be considered as price words. The following filters are applied:

- bounding box height is within a specific range (this range needed to be specified by hand and is store dependent)
- locating the area specified by the bounding box in the original image and verifying the presence of a store-specific price background color (a minimum percentage must be filled by this color)
- word matches a designated price regular expression

Detection of amount words is very similar. They also have font sizes within a specific range, and though the background color is not always consistent, the text color is either black or white, allowing for the application of a color filter. Amounts must also match a regular expression in the format "[number] [unit]".

The following filters are applied to recognize product names:

- bounding box height is within a specific range
- black or white color filter
- store-specific text format filter (e.g., Lidl uses capital letters for product names, while other shops typically use lowercase letters with an initial capital letter)
- texts do not contain promotional words frequently found in flyers

The API-provided block grouping is used to group words that passed the corresponding filter. A block not only groups words with similar font sizes but also those that are related according to the OCR. However, as the API-provided grouping is not entirely accurate, additional postprocessing separates grouped words that are far apart (beyond a predetermined maximum distance) into distinct groups and groups ROIs that are close to each other (within a predetermined maximum distance).

3.2.4 Pairing product name with a price

A simple solution to pair product names with prices is to find the closest price to each product name. While this approach may work in many cases, there are instances where the closest price to a product name corresponds to a different product or instances where multiple prices are located at a similar distance from a product name. Such a situation is depicted in Figure 3.4. Moreover, there may be more product names than prices to pair if a text that is not a product name passes through the product name extraction filters.

Figure 3.4: Problem with closest neighbor pairing in dense flyers – closest neighbouring price to a product name might belong to another product name, e.g. the product "Jar Prostředek na nádobí" has similar distance to the price 299 and 139.90 and the closest neighbour pairing might be incorrect (flyer source: `www.albert.cz`)

To address these challenges, a more robust solution is required. When two or more prices are close to the same product name, there are other names in the flyer corresponding to the rest of those price(s). Rather than minimizing the distance between each pair separately, the matching heuristic should also take the other pairs into account. This can be achieved by minimizing the sum of distances between all pairs. This approach can be formulated as an Integer linear programming (ILP) problem:

$$
\begin{aligned}
\min \sum_{p\in 1\ldots P} \sum_{n\in 1\ldots N} x_{p,n} c_{p,n} \\
\forall p \sum_{n\in 1\ldots N} x_{p,n} \leq 1 \\
\forall n \sum_{p\in 1\ldots P} x_{p,n} \leq 1 \\
\sum_{p\in 1\ldots P} \sum_{n\in 1\ldots N} x_{p,n} = \min(P, N) \\
\text{variables:} \quad x_{p\in 1\ldots P, n\in 1\ldots N} \in \{0,1\}, \\
\text{parameters:} \quad c_{p\in 1\ldots P, n\in 1\ldots N} \in \mathbb{R}^{+},
\end{aligned}
\tag{3.1}
$$

where $x_{p,n}$ is 1 if price p is paired with product name n and $c_{p,n}$ is a cost for pairing price p with product name n (e.g. Euclidean distance in the flyer). The first constraint ensures that each price is paired with only one product name. The second constraint ensures that each product name is paired with only one price. The last constraint ensures that as many pairs as possible are made.

Moreover, when $P = N$, this issue can be treated as a bipartite weighted matching problem, as defined in [1, Chapter 12.4]. This problem does not need to be solved as an ILP task and can be formulated as a minimum cost flow problem [1, Chapter 9][7, Chapter 9], which can be solved in polynomial time using, for example, the Cycle Canceling algorithm [1, Chapter 9.6][7, Chapter 9.3][9].

To formulate this problem as a minimum cost flow problem, a source node with a balance equal to the maximum number of pairs that can be made ($b_s = \min(P, N)$) is created, along with a sink node with a balance of $b_t = -b_s$, and a node for each price and each name with a balance equal to 0. The following edges are added:

- edges from the source node to each price node with zero cost and an upper bound of 1
- edges from each price node to each name node with a cost equal to $c_{p,n}$ and an upper bound of 1
- edges from each name node to the sink node with zero cost and an upper bound of 1

The resulting graph is illustrated in Figure 3.5.

Then if there is a flow equal to 1 from i-th price node to j-th name node in the optimal solution of the minimum cost flow problem, i-th price should be paired with j-th name, otherwise they should not be paired.

This solution works for $P = N$, but the OCR may miss prices or product names or detect extra ones. The optimization will fail if $P \neq N$. In addition, it will enforce creating $P = N$ pairs even if the prices and names are far from

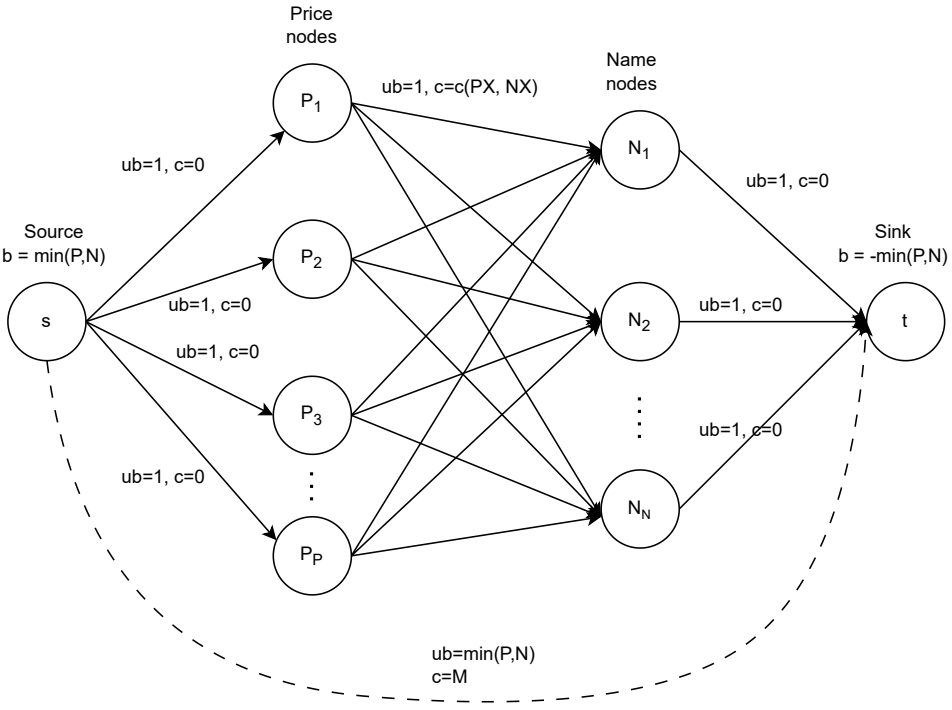


Figure 3.5: Formulation of pairing as a minimum cost flow problem, the dashed flow is optional (one of the solutions for case when $P \neq N$)

each other and not related. One of the following two approaches can be used to solve this.

The first solution is to add one extra edge (dashed flow in Figure 3.5) from the source node to the sink node with an upper bound equal to $\min(P, N)$ and cost equal to a manually set up maximum allowed distance M between name and price. As a result, the problem is always feasible and the optimization only produces pairs with a maximum distance M between name and price, because the flow through the extra edge will be preferred over any pairing with cost $> M$.

The second solution aims to create the maximum number of pairs possible with distances less than the specified maximum distance, denoted as M. The first step is to remove all edges with a cost greater than M. Next, the maximum flow optimizer is executed to determine the maximum flow from the source node to the sink node. The resulting maximum flow corresponds to the maximum number of pairs that can be formed. After that, the minimum cost flow algorithm can be triggered with the source node balance (b_s) set equal to the maximum flow and the sink node balance $b_t = -b_s$.

In the end, the second approach was used because it is more robust, especially in combination with penalization discussed in Section 3.2.4.

To solve the minimum cost flow problem and the maximum flow problem, the OR-tools library [14] is used.

An example of a pairing proposed by the resulting pairing algorithm is shown in Figure 3.6.

Figure 3.6: Result of pairing algorithm, green bounding boxes show detected prices, red bounding boxes show detected names and blue lines show pairing proposed by the pairing algorithm (flyer source: `www.albert.cz`)

Distance (pairing cost) computation

Rather than relying solely on the Euclidean distance as the pairing cost, slight modifications were made to improve the pairing accuracy. Specifically, the length of the portion of the shortest connecting line that traverses any text areas detected by the OCR (multiplied by a constant factor $k \in [0, 1]$) is subtracted from the Euclidean distance. The motivation for this was, that it often happens that the product name is above its price and in between is an extra text regarding the product (this can be seen for example in Figure 3.6 when looking at product "Albert dětské plenky" - the blue shortest connecting line passes through text area).

Additionally, one other modification was made to the pairing cost computation to help to prevent texts present in product images from being mistaken for product names. Using the OpenCV library [17] for edge and contour detection, an attempt was made to localize product images. The detection process based on background extraction approach (described for example in [11]) consists of the following operations:

1. Blurring the image using Gaussian blur with a kernel size of 15×15 pixels to reduce noise.
2. Applying Canny edge detection [2].

[17] `https://opencv.org/`

3. Performing dilation and erosion on the resulting edges (using a kernel size of 5×5 pixels) to merge close edges.

4. Finding contours within the edges.

5. Selecting only contours with area sizes within a specified range. This range is designed to include product images while excluding contours that cover the entire image or are as small as individual text characters.

6. Combining the filtered contour areas into a bitmap. This bitmap can be used to determine whether a product name candidate lies within a product image area.

An example of detected product image areas can be observed in Figure 3.7.

Figure 3.7: Product image detection, from the left to right - a) original image, b) detected product images, c) both images combined (flyer source: `www.lidl.cz`)

Due to inaccuracies, excluding names within detected product image areas might eliminate valid product names. Instead, such names were only penalized – by increasing their distance c to each price as follows:

$$c = M - 1 + \frac{c_{orig}}{M}$$

, where M represents the maximum allowed distance and c_{orig} is the distance before this modification. Consequently, product names identified outside of image areas with a distance $\leq M - 1$ to a price are always prioritized in the optimization process over names situated within image areas. At the same time, the penalized items are still comparable by distance (information about distance is preserved by the $\frac{c_{orig}}{M}$) and $c > M \iff c_{orig} > M$.

Since the product name candidates positioned within detected product image areas are not entirely eliminated and are only penalized, it is not crucial for the detection to be entirely accurate. However, the higher the accuracy, the more helpful it becomes.

3.2.5 Pairing amount to a product

The amount is provided either just below the product name or just above the product price depending on the store. So there is a specific area where the amount has to be. If it's not found there, it is omitted, if there are more detected amounts in that area, the closest one is paired.

3.2.6 Structured PDFs

Detection of product names, prices, and amounts in structured PDFs is easier because exact font type and size can be extracted from the PDF and used to decide whether the text is name, price, amount, or something else. In this case, even the original non-sale price can be extracted from the flyer even though it is crossed, because there is no thread of character recognition error like when using OCR.

3.2.7 Start and end date extraction

Start and end dates are the only relation between pages that needed to be extracted. Some flyer pages list offers that are only available for a part of the week. This is the case with Lidl flyers. On top of the page can be provided information that the offers on this page and the following pages are limited to some date range. So whenever such information is provided it needs to be applied to all following pages, until a new date is provided.

3.2.8 Results

The proposed extraction method was applied to Lidl, Albert, and Billa flyers. To test the extraction accuracy, over 100 offers for each of the three shops were manually reviewed and the extracted results were compared with the real offers presented in the flyers. The correctness of the amount, name, price, and their mutual matching was verified. Results of the comparison are summarized in the table 3.1. As can be seen from the table, the accuracy on the test dataset is 92 % for Lidl flyers, 90% for Albert flyers, and 100% for Billa flyers. Let's discuss why are there differences between the accuracies.

Albert

Albert flyers are dense - the products are listed close to each other and therefore, there is a higher chance that the product name to price matching will be incorrect. Also since the different texts are close to each other, the OCR accuracy is worse and it is more likely that it will group unrelated words into the same paragraph or block.

One advantage of Albert flyers is that the amounts are positioned beneath the product names, which are more challenging to identify than prices. This property can be utilized to enhance product name detection - texts that do not have their corresponding amount can be removed from the list of name

	Lidl	Albert	Billa
Real number of products in the flyer	127	123	111
Correctly extracted products	118	111	111
Number of extracted products	125	112	111
Missed products	2	11	0
Missing amount	2	0	0
Incorrect name - price matching	0	1	0
Incorrectly detected amount	2	0	0
Incorrectly detected price	0	0	0
Incorrectly detected name (incomplete name, name from product image, ...)	3	0	0

Table 3.1: Flyer extraction results

candidates. This approach effectively filters out the majority of non-product name strings from being mistaken as product names. It also explains the higher number of missed products, since both amount and name have to be detected in order for the product to be listed in the results.

Lidl

Products in Lidl flyers are listed further from each other. Therefore the OCR performance is better and the risk of incorrect matching is lower. On the other hand, the amounts are positioned above prices and cannot be used to improve product name detection. This causes a higher number of errors as sometimes words detected in product photos are considered to be product names. Also, the online version of the flyer is only available in worse quality than the Albert flyer, which also has an effect on OCR performance and increases the risk of errors.

Billa

Billa flyers are available as structured PDF files and the text can be extracted directly from them without the need to use OCR. Therefore there are no errors caused by OCR inaccuracies. Also, it is easier to determine which texts are part of the product name, price, or amount based on the exact font and font size provided in the PDF. As a result, the extraction accuracy for the test dataset was 100 %.

Discussion

Missing items are not as critical as incorrectly detected information. Especially incorrect price detection is an issue as it can interfere with the whole price optimization process and a non-cheapest shop combination can be recommended to the user instead of an optimal combination.

Therefore it is crucial to correctly pair the product name with its price. In the test dataset, the pairing failed once. After analyzing the failure, it

was found that it occurred because a product name was not detected, and its corresponding price was the closest price to another product name, leading to an incorrect pairing. Detecting such situations is challenging. The solution used in this work tries to detect such situations when the product is being added to the database. If the product already exists in the database, its lowest (L) and highest (H) prices present in the database are determined. If the detected price deviates from the $[L, H]$ range by more than P percent[18], the offer is not added to the database.

The analysis also revealed cases where the product name was detected incorrectly. The name was incorrect either because the detected name was incomplete or because the text from the product image was detected instead of the product name. These cases are mostly mitigated by word grouping (described in Section 3.2.3) and product image masking (Figure 3.7) and occurred only for three product names.

A more robust product image detection and text merging method [8] would be required to further improve this aspect. However, in most cases, a product with an incorrect name added to the database will have only this one offer assigned to it, and therefore, it will not be prioritized in the search results (Section 2.1.5). Furthermore, the words that must appear in a product name for it to be added to a group closely specify the product type the group represents. Therefore, if essential words are missed, the product is unlikely to be assigned to any group, and thus, it will not impact the process of price optimization even if the shopping list contains product groups.

Incorrect amount detection occurred twice in the test dataset. In both cases, it was due to OCR misdetection. Font sizes of amounts in flyers are small and therefore challenging for the OCR to detect.

To prevent populating the database with incorrect names and amounts, automatic creation of new product records in the database by extractors from PDF flyers can be disabled. Consequently, if the detected product name and amount do not match any existing product (Section 3.3), they are not added to the database. Instead, they are added to a list of products to be reviewed and added manually.

Although there is a room for further improvements, the results are already quite satisfactory (with more than 90% accuracy) considering that flyer extraction is not the main focus of this thesis.

3.3 Product Matching and Merging

When dealing with products from different supermarkets, it is common to encounter products with different names even though they are essentially the same. To effectively compare and merge these products, matching mechanism must be implemented. This section provides an overview of the product matching and merging process employed in the application.

[18] P is a preset constant

3.3.1 Unified Categorization

Each supermarket classifies its products into categories. These categorizations can differ across supermarkets, necessitating the creation of a unified categorization system. A manually created mapping table is used to map each supermarket's categorization to the unified categorization schema. This categorization system has multiple levels, such as "Dairy products" → "Yogurts" → "White yogurts".

3.3.2 Computing Match Rate

When adding a new product to the database, the match rate between the new product and each existing product is computed. If the match rate is higher than a preset threshold, the new product is merged with the existing product that has the highest match rate. This helps to avoid matching unrelated products. The match rate is determined as follows:

1. If the two products have the same barcode, they are always matched.
2. If the packaging of the two products differs, they are considered different products.
3. If the prices of the two products differ by more than P percent[19], they are considered different products.
4. If the products are not in the same category and don't have a common categorization path (e.g., a product in "Dairy products" can be merged with a product from "Dairy products"→"Yogurts", but not with a product from "Fruits and vegetables"), they are considered as different products.
5. Otherwise, the match rate is calculated as

$$0.7 \times \text{nameMatch} + 0.3 \times \text{brandMatch},$$

where nameMatch and brandMatch are the similarities of product names and brand names, respectively. The name similarity must exceed a specified threshold. Additionally, the brand match rate is also computed against the product name. If this rate surpasses the match rate against the other brand name, it is used instead. The motivation for this step is that often the brand name is contained in the product name.

String similarity computation

A set of metrics for product name matching are compared in study by Horch, Kett and Weisbecker [6]. The study proposes and compares different similarity values (metrics) of product names. According to the comparison results, the "Intersection of Words" similarity value produces the best precision with

[19] preset constant

reasonable recall for matching. This means that it produces a low number of false positive matches while detecting a reasonable number of true positives.

The Intersection of Words metric is calculated by dividing the number of shared words between two strings by the total number of words in the longer string. It can be represented by the following formula:

$$\frac{\text{common}(\text{words}(\text{string1}), \text{words}(\text{string2}))}{\max(\text{len}(\text{words}(\text{string1})),\ \text{len}(\text{words}(\text{string2})))}$$

One main issue with this metric is that similar but not identical words (with missing characters, plural vs singular, etc.) don't increase the match rate. Additionally, if the number of words differs between the two strings, the match rate decreases. This lowers the recall when one shop provides longer product names with detailed information and extra marketing words, while another shop provides brief product names. To address these issues, I made two modifications to the metric.

The first modification aims to increase the match rate of names containing words that are not identical but are similar. The study's proposed "Similarity of Strings" metric is used to increase the match rate when words are similar. The pseudocode in Listing 3.2 (lines 7-20) explains the change: it increments the common words count by the "Similarity of Strings" of similar words. The "Similarity of Strings" is a metric computed by comparing two strings using the difflib library's SequenceMatcher ratio method [20]. It is computed using the Longest Contiguous Subsequence (LCS) method as follows:

$$\text{matchRatio} = \frac{2 \times \text{LCS}(\text{word1},\ \text{word2})}{|\text{word1}| + |\text{word2}|}$$

The second modification increases the match rate when the length of strings differs but the longer one of the two strings contains all the words present in the shorter one. Instead of dividing the common word count by the length of the longer string, it is divided by the average length of the two strings. The resulting pseudocode in Listing 3.2 summarizes the modified logic used to compute the name similarity.

[20] `https://docs.python.org/3/library/difflib.html`

Listing 3.2: String similarity computation

```
function similaritiesOfWords(words1, words2){
    maxLength = max(words1.size, words2.size);
    minLength = min(words1.size, words2.size);
    commonWords = numberOfWordsPresentInBothSets(
        words1, words2);

    for (let word1 of words1 \ commonWords){
        bestRatio = 0
        //find most similar word to word1
        for (let word2 of words2 \ commonWords){
            bestRatio = max(
                bestRatio,
                similarityOfStrings(word1, word2)
            )
        }
        if (bestRatio > 0.5){
            commonWords += bestRatio;
            words2.delete(bestRatioWord);
        }
    }

    if (commonWords == minLength)
        return commonWords/((maxLength+minLength)/2);
    else return commonWords/maxLength;
}
```

3.3.3 Merge Process

Once the match rate has been computed and found higher than the preset threshold, the new product is merged with the existing product that has the highest match rate. This process involves updating the product offers.

The old offers from the same shop with unknown offer end date are invalidated and the new offers are added. Invalidation of nondiscount offers is skipped when discount offer alone is being added (this is the case when offer found by flyer extraction is being added).

3.3.4 Challenges and Improvements

The product matching and merging process described above is not without its challenges. The match rate threshold is set to a relatively high value to avoid merging unrelated products. As a result, if the names corresponding to the same product differ too much, they are not matched. To further improve the matching process, perhaps machine learning could be used.

Chapter 4

Conclusion

The primary goals of this work have been successfully achieved, resulting in a comprehensive and user-friendly solution for optimizing shopping lists and minimizing costs.

The designed flyer extraction system is capable of correctly extracting over 90 % of all offers present in a marketing flyer. By reverse-engineering the APIs of multiple online shopping services, specialized scrapers were created to acquire product information from these sources. Methods for matching corresponding products and grouping similar ones were developed. Altogether, this resulted in a comprehensive and up-to-date database, containing over 40,000 offers and 25,000 products from the seven largest supermarket chains operating in the Czech Republic.

Building on this extensive database, a web application was developed that allows users to manage and optimize their shopping lists. It allows comparing the total price of their shopping across different shops and shop combinations.

4.1 Future work

While the goals of this work were achieved, several ideas for future work could further enhance the application and user experience.

Improvements in product matching and grouping could be made. The current product matching process is rigorous to avoid matching of unrelated products. However, this strictness occasionally results in related products not being recognized as such, causing them to remain as separate records in the database. This poses a problem for users seeking specific products. When a user adds one of these products to their shopping list, the optimization process will overlook offers linked to other, separate product records. Consequently, the resulting shopping suggestion might not be optimal, or the product might appear to be unavailable in some shops, as the relevant offers are associated with the other records. This challenge could be addressed by employing more sophisticated product matching tools, such as Natural Language Processing techniques.

Moreover, the product offer database for some shops is incomplete, necessitating manual intervention. For instance, the Kaufland scraper can only acquire offers for products with known barcodes (Section 3.1.6). Currently

only barcodes for products offered in Globus are known, other ones need to be manually collected. Similarly, the Lidl product database is incomplete. It is only limited to discounted products extracted from flyers since Lidl does not provide an online service with database of offered products.

Future work could also include the development of native mobile applications for iOS and Android platforms, making the tool even more user-friendly and accessible. Additionally, introducing features such as prepared shopping lists for different occasions or meals could help users plan their shopping more conveniently. Moreover, adding support for multi-user collaboration would enable users to share and collaborate on shopping lists with friends and family members.

These enhancements would further enhance the application, becoming an even more valuable resource for users looking to optimize their grocery shopping expenses.

The remaining challenge is to maintain the database and keep the data scrapers updated. Shops frequently change product prices, assortment, their online services, and the structure of discount flyers. It is essential to track these changes and adjust the data extractors accordingly.

Glossary

API Application Programming Interface. 2, 19–26, 29, 41

CA Certificate Authority. 20, 22

DAO Data Access Object. 3

DTO Data Transfer Object. 4

HTML Hypertext Markup Language. 21, 24

HTTPS Hypertext Transfer Protocol Secure. 10, 20, 22

ILP Integer linear programming. vii, 1, 16, 18, 30, 31

JSON JavaScript Object Notation. 21

MITM Man in the middle attack. 20, 22–24

OCR Optical character recognition. vii, ix, 2, 26–29, 31, 33, 35–37

PDF Portable Document Format. vii, 26, 35–37

REST Representational state transfer. vii, 3, 4, 9

ROI Region of interest. vii, 27, 29

SMTP Simple Mail Transfer Protocol. 8

SQL Structured query language. 4, 8

UML Unified Modeling Language. ix, 4, 6, 8

URL Uniform Resource Locator. 8, 21

Bibliography

[1] Ravindra Ahuja, Thomas Magnanti, and James Orlin. Network flows. 01 1993.

[2] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

[3] James William Cooper. *Java design patterns: a tutorial*. Addison-Wesley Professional, 2000.

[4] Ignazio Gallo, Alessandro Zamberletti, and Lucia Noce. Content extraction from marketing flyers. volume 9256, pages 325–336, 09 2015.

[5] Dick Hardt. The oauth 2.0 authorization framework. Technical report, 2012.

[6] Andrea Horch, Holger Kett, and Anette Weisbecker. Matching product offers of e-shops. In Huiping Cao, Jinyan Li, and Ruili Wang, editors, *Trends and Applications in Knowledge Discovery and Data Mining*, pages 248–259, Cham, 2016. Springer International Publishing.

[7] B. H. Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer-Verlag, 2012.

[8] Harlinton Palacios Mosquera and Yakup Genç. Recognition and classifying sales flyers using semi-supervised learning. In *2019 4th International Conference on Computer Science and Engineering (UBMK)*, pages 1–6, 2019.

[9] Course of combinatorial optimization. `https://cw.fel.cvut.cz/wiki/courses/ko/start`. (last visited on 05.05.2023).

[10] Flyer data extraction. https://github.com/grahamhoyes/flyer-data-extraction. (last visited on 05.05.2023).

[11] Background removal with python. `https://towardsdatascience.com/background-removal-with-python-b61671d1508a`. (last visited on 25.05.2023).

[12] Google cloud vision ocr. https://cloud.google.com/vision/docs/ocr. (last visited on 05.05.2023).

[13] lp_solve. https://lpsolve.sourceforge.net/5.5/. (last visited on 05.05.2023).

[14] Minimum cost flows solver. https://developers.google.com/optimization/flow/mincostflow. (last visited on 05.05.2023).

[15] Tesseract. https://github.com/tesseract-ocr/tesseract. (last visited on 05.05.2023).

[16] Tesseract documentation - improving the quality of the output. https://tesseract-ocr.github.io/tessdoc/ImproveQuality.html#binarisation. (last visited on 05.05.2023).

[17] Mitmproxy. https://mitmproxy.org/. (last visited on 05.05.2023).

[18] Dinesh Rajput. *Spring 5 Design Patterns: Master efficient application development with patterns such as proxy, singleton, the template method, and more*. Packt Publishing Ltd, 2017.

[19] Ashutosh Satapathy, Jenila Livingston, et al. A comprehensive survey on ssl/tls and their vulnerabilities. *International Journal of Computer Applications*, 153(5):31–38, 2016.

[20] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

Chapter 5

Attachments

5.1 REST API Documentation

5.1.1 Endpoints

- **POST /rest/users**
  Registers a new user.
  Request Body: RegisterDTO (JSON)
- **GET /rest/users/verifyEmail**
  Verifies an email address.
  Request Params: email (String), id (String)
- **GET /rest/users/requestPasswordResetEmail**
  Requests a password reset email.
  Request Params: email (String)
- **GET /rest/users/requestVerificationEmail**
  Requests a verification email.
  Request Params: email (String)
- **POST /rest/users/resetPassword**
  Resets a user's password.
  Request Body: ResetPasswordDTO (JSON)
- **POST /rest/users/changePassword** (ADMIN, USER)
  Changes a user's password.
  Request Body: ChangePasswordDTO (JSON)
- **GET /rest/users/current** (ADMIN, USER)
  Retrieves the current user's data.
  Response: ClientDTO (JSON)
- **DELETE /rest/users/removeShop/{shopId}** (ADMIN, USER)
  Removes a shop from the user's list of preferred shops.
  Path Params: shopId (Integer)

- **POST /rest/users/addShop/{shopId}** (ADMIN, USER)
  Adds a shop to the user's list of preferred shops.
  Path Params: shopId (Integer)

- **DELETE /rest/users/removeMembership/{membershipId}** (ADMIN, USER)
  Removes a membership from the user's membership list.
  Path Params: membershipId (Integer)

- **POST /rest/users/addMembership/{membershipId}** (ADMIN, USER)
  Adds a membership to the user's membership list.
  Path Params: membershipId (Integer)

- **DELETE /rest/users/blacklist/{productId}** (ADMIN, USER)
  Removes a blacklisted product from the user's blacklist.
  Path Params: productId (Integer)

- **POST /rest/users/blacklist/{productId}** (ADMIN, USER)
  Adds a blacklisted product to the user's blacklist.
  Path Params: productId (Integer)

- **PUT /rest/users/setMaxShopCount/{maxShopCount}** (ADMIN, USER)
  Sets the maximum number of shops used optimization for the user.
  Path Params: maxShopCount (Integer)

- **POST /rest/optimize** (ADMIN, USER)
  Optimizes the shopping list for the current user.
  Response: List of ShoppingDTO (JSON)

- **GET /rest/products/search**
  Searches for products by name.
  Request Params: searchString (String)
  Response: List of ProductSearchDTO (JSON)

- **GET /rest/products/id** (ADMIN, USER)
  Retrieves product details by ID.
  Path Params: id (Integer)
  Response: ProductDTO (JSON)

- **GET /rest/products/getRecommendations** (ADMIN, USER)
  Retrieves recommendations of product groups on sale.
  Response: List of ProductSearchDTO (JSON)

- **GET /rest/shops**
  Retrieves all available shops.
  Response: List of ShopDTO (JSON)

- **GET /rest/shopping** (ADMIN, USER)
  Retrieves the current user's shoppings.
  Response: List of ShoppingDTO (JSON, without items)
- **PUT /rest/shoppingItem/itemId/uncheck** (ADMIN, USER)
  Unchecks a shopping item.
  Path Params: itemId (Long)
  Response: No Content
- **PUT /rest/shoppingItem/itemId/check** (ADMIN, USER)
  Checks a shopping item as bought.
  Path Params: itemId (Long)
  Response: No Content
- **PUT /rest/shopping/shoppingId/finish** (ADMIN, USER)
  Finishes a shopping.
  Path Params: shoppingId (Long)
  Response: List of ShoppingListItemDTO (JSON)
- **GET /rest/shopping/shoppingId** (ADMIN, USER)
  Retrieves a shopping by ID.
  Path Params: shoppingId (Long)
  Response: ShoppingDTO (JSON)
- **POST /rest/shopping/start** (ADMIN, USER)
  Starts a new shopping.
  Request Body: ShoppingDTO (JSON)
  Response: ShoppingDTO (JSON)
- **GET /rest/shoppingList** (ADMIN, USER)
  Retrieves the current user's shopping list items.
  Response: List of ShoppingListItemDTO (JSON)
- **PUT /rest/shoppingList** (ADMIN, USER)
  Updates a shopping list item.
  Request Body: ShoppingListItemDTO (JSON)
  Response: ShoppingListItemDTO (JSON)
- **POST /rest/shoppingList** (ADMIN, USER)
  Adds a shopping list item.
  Request Body: ShoppingListItemDTO (JSON)
  Response: ShoppingListItemDTO (JSON)
- **POST /rest/shoppingList/changeItemState** (ADMIN, USER)
  Changes the state of a shopping list item.
  Request Body: ChangeStateDTO (JSON)
  Response: No Content
- **PUT /rest/adminTools/refreshProductDb** (ADMIN)
  Refreshes the product database (needs to be called after database changes

from outside of the application by scrapers.
Response: No Content

- **PUT /rest/adminTools/generateCsvWithAllProducts** (ADMIN)
  Generates a CSV file with all products.
  Response: No Content

5.1.2 Data Transfer Objects (DTOs)

ChangePasswordDTO

- **email**: String
  The user's email address.
- **password**: String
  The new password for the user.
- **oldPassword**: String
  The user's current password.

ChangeStateDTO

- **shoppingListItemId**: Long
  The unique identifier of the shopping list item.
- **state**: String
  The new state of the shopping list item (ACT, NACT, HID, OS).

ClientDTO

- **email**: String
  The email address of the client.
- **firstName**: String
  The first name of the client.
- **lastName**: String
  The last name of the client.
- **maxShopCount**: Integer
  The maximum number of shops the client wants to visit.
- **role**: String
  The role of the client in the system (ADMIN, USER, TO_VERIFY).
- **gender**: String
  The gender of the client (MALE, FEMALE, OTHER).
- **birthDate**: LocalDate
  The birth date of the client.

- **town**: String
  The town where the client resides.
- **shoppingListItems**: List<ShoppingListItemDTO>
  The list of shopping list items associated with the client.
- **shoppings**: List<ShoppingDTO>
  The list of shopping instances associated with the client.
- **currentShopping**: ShoppingDTO
  The current shopping instance of the client.
- **memberships**: List<Integer>
  The list of user's membership's IDs.
- **shops**: List<Integer>
  The list of user's preferred shop's IDs.
- **blacklistedProducts**: Set<Integer>
  The set of product IDs blacklisted by the client.

ClientVerification

- **email**: String
  The email address of the client.
- **uuid**: String
  The unique identifier (UUID) associated with the client verification.

OfferDTO

- **shopId**: Integer
  The identifier of the shop where the offer is available.
- **offerId**: Long
  The unique identifier of the offer.
- **amount**: Float
  The amount of product offered.
- **cost**: Float
  The cost of the product in the offer.
- **isSale**: boolean
  Indicates whether the offer is a sale or not.
- **offerEnd**: LocalDateTime
  The date and time when the offer ends.
- **percentageSale**: Integer
  The percentage of the sale, if applicable.

ProductDTO

- **id**: Integer
  The unique identifier of the product.
- **name**: String
  The name of the product.
- **baseAmountUnit**: String
  The unit of base amount of the product (G,KG,ML,L,PKG,M,KS).
- **baseAmount**: Float
  The base amount of the product.
- **type**: String
  The type of the product (P, G).
- **offersByShop**: Map<Integer, OfferDTO>
  A map containing the shop identifier as the key and the corresponding offer details as the value.
- **subproducts**: List<SubproductDTO>
  A list of subproducts associated with the product.
- **packageSizeUnit**: String
  The unit of package size of the product (G,KG,ML,L,PKG,M,KS).
- **packageSize**: Float
  The size of the package of the product.
- **brand**: String
  The brand of the product.
- **category**: String
  The category the product belongs to.

RegisterDTO

- **email**: String
  The email address of the user registering.
- **firstName**: String
  The first name of the user registering.
- **lastName**: String
  The last name of the user registering.
- **password**: String
  The password for the user registering.
- **gender**: String
  The gender of the user registering (MALE,FEMALE,OTHER).

- **birthDate**: String
  The birth date of the user registering, stored as a string in the format "yyyy-MM-dd".
- **town**: String
  The town where the user registering lives in.

ResetPasswordDTO

- **email**: String
  The email address of the user requesting a password reset.
- **password**: String
  The new password for the user.
- **uuid**: String
  The unique identifier for the password reset request.

ShopDTO

- **id**: Integer
  The unique identifier of the shop.
- **name**: String
  The name of the shop.
- **memberships**: List<Membership>
  The list of memberships associated with the shop.

ShoppingDTO

- **id**: Long
  The unique identifier of the shopping.
- **shopIds**: List<Integer>
  The list of unique shop identifiers where the shopping items are bought.
- **timestamp**: LocalDateTime
  The timestamp of the shopping creation.
- **state**: String
  The state of the shopping (ONGOING, FINISHED, PARTIAL, ABORTED, UNSOLVABLE).
- **shoppingItems**: List<ShoppingItemDTO>
  The list of shopping items in the shopping.
- **totalCost**: Float
  The total cost of the shopping.

- **extraCost**: Float
  The cost of shopping items corresponding to products that the user only wanted to buy if some condition was met (sale, price lower than limit) and for which this condition was met.

ShoppingItemDTO

- **id**: Long
  The unique identifier of the shopping item.
- **offerId**: Long
  The unique identifier of the offer associated with the shopping item.
- **confirmed**: Boolean
  The flag indicating whether the shopping item is confirmed (already bought) or not.
- **offerEnd**: LocalDateTime
  The end date and time of the offer associated with the shopping item.
- **percentageSale**: Integer
  The percentage discount of the offer associated with the shopping item.
- **cost**: Float
  The cost of the shopping item.
- **name**: String
  The name of the shopping item.
- **brand**: String
  The brand of the shopping item.
- **packageSizeUnit**: String
  The unit of the package size of the shopping item (G,KG,ML,L,PKG,M,KS).
- **packageSize**: Float
  The package size of the shopping item.
- **amountUnit**: String
  The unit of the amount of the shopping item (G,KG,ML,L,PKG,M,KS). Used for type MISS or EXTRA (when concrete product is not associated with the shopping item).
- **amount**: Float
  Number of pieces.
- **category**: String
  The category of the shopping item.
- **shopName**: String
  The name of the shop where the shopping item is to be bought.

- **membershipName**: String
  The name of the membership associated with the shopping item, if any.
- **productId**: Integer
  The unique identifier of the product associated with the shopping item.
- **type**: String
  The type of the shopping item (MISS, EXTRA, REQ, NOTE).

ShoppingListItemDTO

- **id**: Long
  The unique identifier of the shopping list item.
- **type**: String
  The type of the shopping list item (N, P).
- **amount**: Float
  The amount of the shopping list item.
- **amountUnit**: String
  The unit of the amount of the shopping list item (G,KG,ML,L,PKG,M,KS).
- **maxCost**: Float
  The maximum cost of the shopping list item.
- **state**: String
  The state of the shopping list item (ACT, NACT, HID, OS).
- **lastBuyDate**: LocalDate
  The date when the shopping list item was last bought.
- **previousPurchasesCount**: Integer
  The number of times the shopping list item was previously purchased.
- **productId**: Integer
  The unique identifier of the product associated with the shopping list item.
- **category**: String
  The category of the shopping list item.
- **name**: String
  The name of the shopping list item.
- **isGroup**: boolean
  The flag indicating whether the shopping list item refers to a product group or not.
- **bestOffer**: OfferDTO
  The best offer available for the shopping list item.

- **shopIds**: List<Integer>
  The list of unique shop identifiers where the shopping list item is available.
- **subproducts**: List<SubproductDTO>
  The list of subproducts associated with the shopping list item.

SubproductDTO

- **id**: Integer
  The unique identifier of the subproduct.
- **name**: String
  The name of the subproduct.
- **packageSizeUnit**: String
  The unit of the package size of the subproduct (G,KG,ML,L,PKG,M,KS).
- **packageSize**: Float
  The package size of the subproduct.
- **brand**: String
  The brand of the subproduct.
- **bestOffer**: OfferDTO
  The best offer available for the subproduct.
- **shopIds**: List<Integer>
  The list of unique shop identifiers where the subproduct is available.

5.2 Albert API

In the response formats, irrelevant attributes are omitted for clarity.

5.2.1 Common for all requests

Base URL	https://api.albertdomuzdarma.cz/
Method	GET

GET parameters	operationName= //depends on request
	variables={
	//depends on request
	}
	extensions={
	"persistedQuery": {
	"version": 1,
	"sha256Hash": //depends on request
	}
	}

Response format	JSON

5.2.2 Acquisition of list of all categories

GET parameters	operationName=LeftHandNavigationBar
	variables={
	"rootCategoryCode": "",
	"cutOffLevel": "4",
	"lang": "cs"
	}

Response format	{"data": {"leftHandNavigationBar": {
	"categoryTreeList": [{
	"categoriesInfo": [{
	"categoryCode": "zeSILF",
	"levelInfo": [{
	"name": "Grilovaci tacky",
	"productCount": 1,
	"url": "/cs-cz/shop/Domacnost-a-zahrada/
	Zahrada/Grilovani/Grilovaci-tacky/c/zeSILF",
	"code": "zeSILF",
	"__typename": "CategoryLevelInfo"
	}],
	"__typename": "CategoryInfo"
	}],
	"level": "4",
	"__typename": "CategoryLevel"
	}],
	"__typename": "LeftHandNavigationBar"
	}}}

5.2.3 Retrieve a product page in a category by category code

GET parameters	operationName=GetCategoryProductSearch
	variables={
	"category": "CATEGORY_ID",
	"searchQuery": "",
	"lang": "cs",
	"pageNumber":0,
	"pageSize":20,
	"filterFlag":true
	}

Response format	{"data": {"categoryProductSearch": {"products": [
	{
	"available": true,
	"code": "24228862",
	"badgeBrand": {
	"code": "albertova trznice",
	"image": {
	"url": "/medias/sys_master/hb4/hb2/8834760474654.png",
	"___typename": "Image"
	},
	"___typename": "ProductBadge"
	},
	"images": [
	{
	"format": "thumbnail",
	"imageType": "PRIMARY",
	"url": "/medias/sys_master/h4e/h73/8816931635230.jpg",
	"___typename": "Image"
	},
	],
	"name": "Albert Paprika sladka",
	"onlineExclusive": null,
	"price": {
	"approximatePriceSymbol": "cca",
	"currencySymbol": "Kc",
	"formattedValue": "36,90 Kc",
	"priceType": "BUY",
	"supplementaryPriceLabel1": "1 kus = 9,95 Kc",
	"supplementaryPriceLabel2": "2 ks",
	"showStrikethroughPrice": true,
	"discountedPriceFormatted": "19,90 Kc",
	"discountedUnitPriceFormatted": "5 kus = 9,37 Kc",
	"unit": "piece",
	"unitPriceFormatted": "36,90 ",
	"unitCode": "pieces",
	"unitPrice": 36.9,
	"value": 36.9,
	"___typename": "Price"
	},
	"productProposedPackaging": 1,
	"productProposedPackaging2": null,
	"url": "/cs-cz/shop/Ovoce-a-zelenina/Zelenina/
	Plodova-zelenina/Albert-Paprika-sladka/p/24228862",
	"___typename": "Product"
	},
	"___typename": "CategoryProductSearchList"
	}}}

5.2.4 Getting a specific product based on ID

GET parameters	operationName=ProductDetails variables={"productCode":"PRODUCT_CODE","lang":"cs"}

Response format	{"data": {"productDetails": {
	"available": true,
	"badgeBrand": {"code": "albertova trznice"},
	"categories": [
	{"code": "zeGG10","name": "Jablka a hrusky"},
	{"code": "zeGG01","name": "Ovoce"},
	{"code": "zeG001","name": "Ovoce a zelenina"}
	],
	"code": "20442422",
	"description": "Stavnate zelene jablka odrudy Granny Smith.",
	"isAvailableByCase": false,
	"limitedAssortment": false,
	"manufacturerName": "",
	"manufacturerSubBrandName": null,
	"mobileClassificationAttributes": [
	{"code": "BRAND","value": "ALBERTOVA TRZNICE"}
	],
	"name": "Jablka Granny Smith skladana",
	"onlineExclusive": null,
	"price": {
	"approximatePriceSymbol": "cca",
	"averageSize": 0.25,
	"currencySymbol": "Kc",
	"discountedPriceFormatted": null,
	"supplementaryPriceLabel1": "1 kg = 44,90 Kc",
	"supplementaryPriceLabel2": "250 g",
	"discountedUnitPriceFormatted": null,
	"unit": "kg",
	"unitPrice": 44.9,
	"value": 11.23,
	"variableStorePrice": false,
	"warehouseCode": "2316",
	},
	"productProposedPackaging": 1,
	"productProposedPackaging2": null,
	"proposedPackagingByPiece": 1,
	"proposedPackagingByCase": null,
	"purchasable": true,
	"stock": {
	"inStock": true,
	"inStockBeforeMaxAdvanceOrderingDate": false,
	"partiallyInStock": false,
	"availableFromDate": null,
	"___typename": "Stock"
	},
	"uid": null,
	"___typename": "Product"
	}}}

5.3 Globus shopping list API

In the response formats, irrelevant attributes are omitted for clarity.

5.3.1 Common elements of requests

Base URL	https://selfscanwebservice1.globus.cz/api
Headers	'Accept: */*', 'Connection: keep-alive', 'User-Agent: GlobusCZ/1.9.2 (cz.globus.mujglobus; build:298; iOS 15.6.0) Alamofire/5.6.2', 'Accept-Language: cs-CZ;q=1.0, en-CZ;q=0.9',
Response format	JSON

5.3.2 Login request

Path	/connect/token
Request type	POST

POST parameters	{
	username: c_up>CCD17A3F-04F0-4D5C-A3F3-A145C9A9EABB>
	iPhone SE>EMAIL,
	password: PASSWORD,
	scope: 'apiCustomerProfile apiCoupon apiOnlineAsset apiProductCatalog
	apiRecommender apiShoppingList apiNotificationRegistration
	apiArticleSnitchService apiGastro apiFoodPickup
	apiCustomerCockpit apiIMProxy offline_access',
	grant_type: "password"
	}

Extra Headers	'Content-Type: application/x-www-form-urlencoded', 'Authorization: Basic HARD_CODED_BEARER_TOKEN_IN_APP',

Response format	{
	"access_token": ACCESS_TOKEN,
	"expires_in": 3600,
	"refresh_token": REFRESH_TOKEN,
	"token_type": "Bearer"
	}

5.3.3 Renew token request

Path	/connect/token
Request type	POST

POST parameters	{
	grant_type: 'refresh_token',
	refresh_token: REFRESH_TOKEN,
	}

Extra Headers	'Content−Type: application/x−www−form−urlencoded', 'Authorization: Basic HARD_CODED_BEARER_TOKEN_IN_APP',

Response format	{
	"access_token": ACCESS_TOKEN,
	"expires_in": 3600,
	"refresh_token": REFRESH_TOKEN,
	"token_type": "Bearer"
	}

5.3.4 Get suggestions (search)

Path	/Recommender/2/houses/4003/suggestions
Request type	GET
Query parameters	"filter": "SearchText:contains TEXT;;ReturnEntities:in Product", "page": 0, "pageSize": 10
Extra Headers	'Content−Type: application/json', 'X−System−Version: 15.6', 'X−App−Platform: iOS', 'X−App−Version: 298',
Response format	[{ "ID": "00174769000", "LastChange": "2023−01−31T22:12:32", "Text": "Zott Zottarella Mozzarella classic 125 g", "Type": "Product", "Vanr": "00174769000" },...]

5.3.5 Get details

Path	/ProductCatalog/2/houses/4003/products
Request type	GET
Query parameters	filter: "vanr:in VANR1,VANR2,VANR3,....", page: 0, pageSize: 100
Extra Headers	'Content−Type: application/json', 'X−System−Version: 15.6', 'X−App−Platform: iOS', 'X−App−Version: 298',

Response format	{
	"allergens": [],
	"articleType": 0,
	"brand": {"brandId": "0","name": "normalni"},
	"contains": "...",
	"ean": [
	"4014500234816",
	"4014500504391",
	],
	"modified": "2023−01−31T21:12:32",
	"name": "Zott Zottarella Mozzarella classic 125 g",
	"nutritionValues": [...]
	"producer": "Zott SE & Co. KG",
	"productCategories": [
	"cls_czr_mozzarella_bryndza",
	"cls_czr_soft_cheeses_mozzarella",
	"cls_czr_cheese",
	"cls_czr_milk_dairy_products_and_eggs",
	],
	"productInHouse": {
	"actualPrice": 36.9,
	"availability": "A",
	"discountPercentage": null,
	"modified": "2023−02−06T05:47:28",
	"originalPrice": 36.9,
	"placements": [
	{
	"category": "Syry",
	"department": "Cerstve potraviny",
	"facing": 3,
	"name": "Syry_Salatove 2.cast_5RJ_03_4P",
	"presentationStock": 27.0,
	"presentationType": 2,
	"subcategory": "Salatove syry"
	}
	],
	"priceValidFrom": "2022−11−21T00:00:00",
	"priceValidTo": "9999−12−31T23:59:59",
	"securityTag": false,
	"stockAmount": 56.0
	},
	"unitAmount": 125.0,
	"unitId": "g",
	"vanr": "00174769000",
	},

5.4 Globus scanner API

In the response formats, irrelevant attributes are omitted for clarity.

5.4.1 Common elements of requests

Base URL	https://selfscanwebservice1.globus.cz/api
Request type	POST
Headers	'Content−Type: application/json; charset=utf−8', 'Accept: ∗/∗', 'Connection: keep−alive', 'X−StoreNumber: SHOP_ID', 'Accept−Language: cs−CZ,cs;q=0.9', 'User−Agent: GlobusCZ/298 CFNetwork/1335.0.3 Darwin/21.6.0'
Response format	JSON

5.4.2 Initiation of shopping

Path	/shopping/start
POST parameters	{ "data": { "appVersion": "", "branchId": SHOP_ID, "clubCard": CLUBCARD_ID, "lang": "", "location": { "lat": 0, "lng": 0 } }, "deviceId": DEVICE_ID }
Response format	{ "address": "Praha − Cerny most", "branchId": 4002, "currency": "Kc", "customerAddressing": "ACCOUNT_NAME", "customerGender": "", "customerId": 1490445, "expire": "2022−10−18T19:54:05.981Z", "goldenScanner": null, "name": "Cerny most", "onlinePayment": false, "showTutorial": true, "token": TOKEN }

5.4.3 Finalization of shopping

Path	/shopping/stop

POST parameters	{
	"data": {
	"reason": "UserCancel"
	},
	"deviceId": DEVICE_ID,
	"token": TOKEN
	}

Response format	{
	"succcess": true
	}

5.4.4 Getting information about an item

Path	/article

POST parameters	{
	"data": {
	"articles": [BARCODE],
	"branchId": SHOP_ID
	},
	"deviceId": DEVICE_ID,
	"token": TOKEN
	}

Response format	{
	"7613035093331": {
	"articleImage": null,
	"articleKey": "7613035093331",
	"articlePrice": 60.0,
	"articlePriceBeforeDiscount": 112.9,
	"departmentId": "627",
	"depositArticle": null,
	"depositKey": "",
	"groupByInnerArticleKey": false,
	"innerArticleKey": null,
	"isDiscountEAN": true,
	"isForSale": true,
	"isWeightedItem": false,
	"maxAmount": 200,
	"multipack": false,
	"name": "NF ORION GRANKO 450G",
	"pnb": "1020771ST",
	"ratio": 1.0,
	"savers": [
	{
	"active": true,
	"articleDependencies": null,
	"articleKey": "7613035093331",
	"articleKeys": [
	"7613035093331"
	],
	"bucketId": 1,
	"clubCard": false,
	"description": "SLEVA",
	"end": "2022−10−17T22:00:00",
	"isIgnored": false,
	"minimumItemValue": 0.0,
	"start": "2022−10−15T22:00:00",
	}
	],
	"secondName": "NF ORION GRANKO 450G",
	"secondaryCodeRequired": false,
	"subDepId": "0",
	"totalPrice": 60.0,
	"units": null,
	"weight": null
	}
	}

5.5 Billa API

In the response formats, irrelevant attributes are omitted for clarity.

5.5.1 Getting page of products belongingto category

URL	https://shop.billa.cz/api/categories/CATEGORY/products?page=PAGE&pageSize=30&sortBy=relevance
Request type	GET

Response format	{
	"count": 1,
	"offset": 0,
	"total": 1,
	"results": [
	{
	"amount": "1",
	"category": "Citrusy",
	"descriptionShort": "Pomeranc",
	"name": "Pomeranc",
	"packageLabel": "",
	"parentCategories": [[
	{"name": "Ovoce a zelenina","slug": "ovoce−a−zelenina−1165"},
	{"name": "Ovoce","slug": "ovoce−1166"},
	{"name": "Citrusy","slug": "citrusy−1167"}
	]],
	"price": {
	"baseUnitShort": "kg",
	"basePriceFactor": "1",
	"regular": {
	"perStandardizedQuantity": 4990,
	"promotionValue": 1646,
	"promotionValuePerStandardizedQuantity": 4990,
	"tags": [],
	"value": 1646
	}
	},
	"productId": "7a3868c4−e7ba−45ba−903c−34f3b35eda65",
	"sku": "82−203616",
	"slug": "pomeranc−82203616",
	"volumeLabelKey": "kg",
	"weight": 1.0,
	"isWeightArticle": true,
	"isWeightPieceArticle": true,
	"weightPerPiece": 330,
	"brandMarketing": "",
	"grossWeight": "",
	"productMarketing": "Odruda Navelina, I. jakost",
	"regulatedProductName": "Pomeranc",
	"minQuantity": 1,
	"quantityStepSize": 1,
	"published": true,
	"seasonal": false,
	"weightArticle": true,
	"weightPieceArticle": true
	}
	]
	}

5.6 Kaufland application API

In the response formats, irrelevant attributes are omitted for clarity.

5.6.1 Access token renewal

URL	https://account.kaufland.com/token-srv/token

POST parameters	{
	client_id: CLIENT_ID,
	grant_type: 'refresh_token',
	refresh_token: REFRESH_TOKEN,
	v: '1.5.0',
	preferredStore: STORE_ID
	}

Headers	'Content−Type: application/x−www−form−urlencoded; charset=utf−8' , 'Connection: Keep−Alive', 'User−Agent: okhttp/4.9.2'
Response format	JSON

5.6.2 Common elements of requests

Base URL	https://live.api.schwarz/kfl/mss/myscan/v1/Basket
Request type	POST
Headers	'Authorization: Bearer ACCESS_TOKEN', 'CidaasID: CIDAAS_ID, 'Kl_Country: CZ', 'Accept: application/json', 'Content−Type: application/json; charset=utf−8', 'Connection: Keep−Alive', 'User−Agent: okhttp/4.9.2',
Response format	JSON

5.6.3 Initiation of shopping

Path	/Start
POST parameters	{ "customerId": CUSTOMER_ID, "language": "CZ", "origin": "Android", "storeId": STORE_ID },

Response format	{
	"attributes": {
	"expires": "99991231235959",
	"storeName": "2050 – Praha–Mecholupy"
	},
	"basketId": "BASKET_ID",
	"basketState": "Started",
	"basketView": {
	"items": [],
	},
	}

Response set-cookie	BusinessServerSessionId: BUSSINESS_SESSION_ID

5.6.4 Remove item from cart

Path	/Remove
Cookie	BusinessServerSessionId: BUSSINESS_SESSION_ID
POST parameters	{ "basketId": BASKET_ID, "barcode": ANY_BARCODE, },
Response format	Code 200

5.6.5 Adding item to cart

Path	/Add
Cookie	BusinessServerSessionId: BUSSINESS_SESSION_ID
POST parameters	{ "basketId": BASKET_ID, "barcode": BARCODE, },
Response format	{ "addedItem": { "barcode": "4337185855528", "id": "P00138023", "longDescription": "KLC.Tortilla.6ks370g", "price": { "grossPrice": 25.9, "itemDiscount": 0.0, "netPrice": 25.9, }, "quantity": 1, "shortDescription": "KLC.Tortilla.6ks370g" }, "attributes": { "expires": "20221024104226" }, "basketId": "yhljmYfJGkmEl7SnFKjY7A", "basketState": "Shopping", "basketView": { "items": [...], } }
Response format if packaging required	{ "availablePackagings": [{ "capacity": 0, "descriptions": [], "type": "None" }, { "capacity": 1, "descriptions": [], "packagingId": "00138023", "type": "Multipack" }], "resultMessage": "PackagingRequired" }