



SILESIAN UNIVERSITY OF TECHNOLOGY IN GLIWICE  
INTERDISCIPLINARY STUDIES IN AUTOMATIC CONTROL  
AND ROBOTICS, ELECTRONICS AND  
TELECOMMUNICATION, INFORMATICS

**Engineering thesis**

Containerization of data warehouse creation and management processes

Author: Karol Latos

Supervisor: dr inż. Anna Gorawska

Gliwice, November 2021





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Scope of the work . . . . .	1
1.2	Thesis contents . . . . .	2
<b>2</b>	<b>Project basis</b>	<b>5</b>
2.1	Data on the internet . . . . .	5
2.2	Python language . . . . .	6
2.2.1	Selenium and Beautiful Soup . . . . .	7
2.2.2	Numpy, Pandas and Scikit-learn . . . . .	7
2.2.3	Matplotlib and Seaborn . . . . .	7
2.3	Docker containers . . . . .	8
2.4	Data warehouses . . . . .	9
2.5	Justification of the thesis topic . . . . .	9
<b>3</b>	<b>Specification</b>	<b>11</b>
3.1	Functional requirements . . . . .	11
3.2	Non-functional requirements . . . . .	12
3.3	Problem domain . . . . .	12
3.3.1	Trading card games . . . . .	13
3.3.2	Market-related entities . . . . .	13
3.3.3	Problem formulation . . . . .	14

3.4	Data warehouse modelling . . . . .	14
3.4.1	Base tables . . . . .	14
3.4.2	Helper views . . . . .	16
3.5	Containers overview . . . . .	16
<b>4</b>	<b>Project architecture</b>	<b>19</b>
4.1	Containers . . . . .	19
4.1.1	Container: <i>firefox_webdriver</i> . . . . .	20
4.1.2	Container: <i>mysql_database</i> . . . . .	20
4.1.3	Container: <i>data_gathering</i> . . . . .	20
4.1.4	Container: <i>database_manager</i> . . . . .	20
4.1.5	Container: <i>data_miner</i> . . . . .	21
4.1.6	Container: <i>node_server</i> . . . . .	21
4.1.7	Container orchestration . . . . .	21
4.2	Services . . . . .	22
4.3	Data pipeline . . . . .	22
4.4	Compatilibity . . . . .	23
<b>5</b>	<b>Data gathering implementation</b>	<b>25</b>
5.1	Program configuration . . . . .	25
5.2	Used services . . . . .	25
5.2.1	Web service . . . . .	25
5.3	Local directories . . . . .	25
5.4	Run scheduling . . . . .	26
5.5	Data pickling and validation . . . . .	26
5.6	Loop over cards . . . . .	26
5.7	Soup decomposition . . . . .	26
5.7.1	Sale offers table . . . . .	26
5.8	Data pipeline output . . . . .	26

<b>6 Database management and data mining</b>	<b>27</b>
6.1 Program configuration . . . . .	27
6.2 Used services . . . . .	27
6.2.1 Database service . . . . .	27
6.3 Run scheduling . . . . .	28
6.4 Tables from staging area . . . . .	28
6.5 Data transformation . . . . .	28
6.6 Program configuration . . . . .	28
6.7 Used services . . . . .	28
6.7.1 Mining service . . . . .	28
6.8 Run scheduling . . . . .	29
6.9 Feature extraction . . . . .	29
<b>7 Results</b>	<b>31</b>
7.1 Testing . . . . .	31
7.2 Version control . . . . .	31
7.3 Project variation in time . . . . .	31
<b>8 Summary</b>	<b>33</b>
8.1 Result assessment . . . . .	33
8.2 Scaling and customizability . . . . .	33
8.3 Conclusions . . . . .	33
<b>Bibliography</b>	<b>35</b>
<b>A Used symbols</b>	<b>37</b>
<b>B Another appendix</b>	<b>39</b>

# Chapter 1

## Introduction

The subject of this engineering thesis is a standalone data warehouse built using multiple Docker containers. The thesis describes how the data is gathered, processed and analyzed, and what components make up each container, as well as their mutual coordination. Additionally, practical results of the project are presented in form of answers to real-world questions, together with steps taken to achieve performance optimization and code robustness.

The basic assumptions of the system are the following:

1. The application is portable, standalone and platform-independent.
2. The application requires the minimal amount of setup in order to run.
3. The code is robust in recovering from errors and able to run continuously.
4. Proper programming practices are followed whenever it is possible.

### 1.1 Scope of the work

The work done in relation to one of the containers in this project is partially based on a group project<sup>1</sup>, concerned with the acquisition of data from a web service in order to answer simple questions with SQL queries. For that project, the author of this thesis implemented the data gathering part using Python language with Selenium module.

---

<sup>1</sup>Group members: Jakub Sieńko, Kacper Garcon and the author, collaborating during an university course *Data Warehouses and Data Mining Systems*

This individual work has been transferred and heavily modified in order to compose a container-based solution, with other containers written by the author from scratch for the purpose of this thesis.

In totality, the following parts were implemented:

- data gathering container,
- database manager container,
- data mining container,
- *nodejs* container with simple web application,
- logging service,
- flag management service,
- database management service,
- web scraping Selenium-based service,
- data management service,
- data mining service.

Additionally, the author researched topics concerned with container orchestration as a simplification of system architecture; Selenium framework for dynamic web data retrieval; optimization of “big data” SQL queries and management of large datasets; full-stack web development in Node.js environment and other related fields.

## 1.2 Thesis contents

Chapter (2) is concerned with the changes in technology leading to various options of exploiting data availability on the Internet, while also deliniating the tools which are at the core of this project. It also explains the motivation for choosing such thesis topic.

Chapter (3) describes the domain of the problem with functional and non-functional requirements of the system. It also gives a high-level view of the data warehouse and the data inside it.



Chapter (4) contains technical information about the coordination of the containers, how they utilize the services and gives an overview of the data pipeline.

Chapters (5) and (6) are centered around the three main containers, taking a deep dive inside each one of them and following the data on its path from the website to market reports.

Chapter (7) reflects on the practical results generated from the data analysis, while contrasting them with initially posed questions. Additionally, the chapter describes the performance of the code under normal and abnormal conditions, and how the project changed with time.

Chapter (8) summarizes the effects achieved during the implementation of the system and formulates conclusions about possibilities of utilizing containerization to create and manage data warehouses.



# Chapter 2

## Project basis

*“Data is the sword of the 21st century,  
those who wield it well, the Samurai.”*

---

— Jonatan Rosenerg, former Senior Vice President  
of Products, Google

In the contemporary world, the Internet facilitates the backbone of world-wide services, including e-commerce, transportation, entertainment and businesses. This relatively recent change presented an open world of possibilities, with new creative ideas emerging every year. In comparison to the reality of the past, the amount of information available at hand is unprecedentedly greater than ever before, constituting an advancement so significant, it can be argued to be as important as the invention of the printing press or even written language. Tapping into this immense ocean of knowledge is possible by utilizing adequate tools, like programming languages capable of producing specifically targeted scripts. This potentially is a fertile ground for automation; and to take advantage of it means to extend one’s capability of processing information, possibly by orders of magnitude, to access previously unavailable knowledge.

### 2.1 Data on the internet

What is the history of internet? What is the state of the internet today? How good is the average bandwidth? How are machines communicating with each other and to what means? Instant availability of information is only practical for computers, need

to automate the gathering and analysis of data. Enter web scraping.

## 2.2 Python language

Python is a high-level general-purpose programming language. It provides levels of abstraction from the machine architecture, so that the user doesn't have to worry about managing memory allocation or typing the variables. It can be used to develop applications in a myriad of domains. Its history begins in December 1989, when a Dutch programmer Guido von Rossum became working on a successor to the ABC language [3]. Released in 1991, with major consecutive versions in 2000 and 2008<sup>1</sup>, it today became a robust language for just about anything, from statistics and modelling, to computer vision and creating web applications. Python is taught in schools as an entry-level language, while at the same time being used by NASA<sup>2</sup>.

Python offers a variety of modules and packages (also called libraries), which are files of code written by other developers with collections of functionalities, for example to manage image files. Among the most popular libraries we have *numpy* — for managing matrices and multi-dimensional tables similar to MATLAB; *openCV* — for processing and analyzing images and videos; *requests* — a simple HTTP library for communicating with the server; and many more.

This project utilizes several crucial libraries, which are described in separate subsections below. The rest is briefly described in the following list:

- *checksumdir*: used for calculating a checksum of given directory, similar to a file checksum. It helps determine whether new data has arrived, is it complete and can it be passed further;
- *time*: used for measuring time between two points in code, sleeping (that is, stopping the execution) and getting information on the current date. Helps with scheduling the run until the day changes and measuring performance of various parts of code;
- *shutil*: used for creating and removing non-empty directories. Helps additionally isolate the shared data inside each container, so that it's invulnerable to changes

---

<sup>1</sup>Python 3, current version.

<sup>2</sup><https://github.com/nasa/podaacpy> is used for crucial communications with Jet Propulsion Laboratory

in the original folder once running;

- *os*: used for managing local files and ensuring proper directory structure on the first run — creating shared folders for storing data, logs and flags.

### 2.2.1 Selenium and BeautifulSoup

These libraries make it possible to connect to any website and emulate user behaviour in an automatic way. Selenium is a framework responsible for creating a webdriver (virtual, in-code browser object) and visiting requested URLs, while looking for specific elements (like buttons for expanding the page) or deciding whether the response from the page is complete or it needs more time to load. BeautifulSoup on the other hand is a simple but powerful system for navigating HTML code. It provides support for finding particular webpage elements like *div* or *span*, by requesting their *class*, *id*, or any other attribute. Every HTML page is converted into a soup object, similar to a nested dictionary, which then can be searched using provided methods.

### 2.2.2 Numpy, Pandas and Scikit-learn

Numpy, briefly mentioned before, is a mathematical library which is a basis for many other libraries, such as Pandas or Scikit-learn. Pandas revolves around Dataframes and Series — two- and one-dimensional data structures similar to Excel or SQL tables. One of the advantages is the speed of processing; selecting tens of thousands rows out of millions, based on some condition, is almost instantaneous. Dataframes come with set of tools for their manipulation, i.e. aggregation functions, joins, concatenation, etc. When this data is passed, scikit-learn is responsible for creating statistical models trying to find correlations, predict outcomes, as well as provide decision support based on the data. It is arguably one of the most important libraries in machine learning domain.

### 2.2.3 Matplotlib and Seaborn

Matplotlib and Seaborn are two libraries for data visualization, where the latter is an extension of the former. Matplotlib gives basic plotting functionalities and proves to be a robust module, which can be used on its own. However, in order to reduce

the amount of code written and standardize the outcomes, Seaborn is used to provide more complex visualizations without an extensive use of Matplotlib.

## 2.3 Docker containers

Containerization (or OS-level virtualization) is a way of isolating resources inside an operating system without using virtual machines (VMs), to create self-enclosed, lightweight executables. The key difference from VMs is that virtualization uses a hypervisor — software that hosts guest operating systems and distributes hardware resources among them. Any process running inside a virtual machine only sees the guest operating system. Meanwhile, containerization uses only the host operating system and a container engine (e.g. *Docker*). From the point of view of a process running inside a container, the directory structure may largely differ from what user sees on the disk. Container should have only the minimal number of libraries and dependencies required to run it. Generally speaking, containerization is a paradigm for the operating system kernel to allow many isolated *user spaces* to exist in a shared environment, which translates to container sharing the filesystem with the host operating system without conflicts. However, recreating the container may mean losing all of our data and starting fresh. To avoid that, methods for data persistence are available, two of which are most popular and used in this project:

- Docker volumes — internal Docker storage, which persists removing the container image if stated explicitly. These volumes are not seen by the host OS.
- Bind mounts — specified directories inside the container, that will correspond to actual directories on the host operating system. This technique is similar to mounting an USB device in a filesystem.

The goal of containerization is to deploy applications securely and fast, without worrying about OS compatibility. The act of abstracting our software from the host operating system allows containers to be portable and stand-alone. Additionally, one can orchestrate many containers to run together and share the OS resources in order to perform a common task. This creates a perfect opportunity to use containerization for an application managing data warehouse, since the data has to go through many different stages in a pipeline, which correspond to separate processes, each inside its own container.

## 2.4 Data warehouses

Data warehouse is a data organization concept that originated in late 1980s in IBM [1]. Barry Devlin and Paul Murphy were trying to find a way to optimize the processing of data from common source to different destinations, called decision support systems. These systems were composed of software taking various data as input, and producing a metric for finding solutions to a given problem. One example is using a decision support system highlighting unusual areas of a brain scan from a MRI, for faster recognition of potentially malicious changes. Before the practice of data warehousing, multiple systems had to acquire data independently from a business source, process it and then perform needed analysis. However, this approach yields several problems, most obvious of which is computational redundancy and consequently wasting of resources.

Devlin and Murphy's idea was to find commonalities between different decision support systems, gather all the needed data at once, process it and then store it in a ready-to-use format. This way, any application could request a specific set of data, tailored for its needs, without the need to perform heavy computation every time. These datasets are called *data marts*, and the isolation of decision support systems from their individual data sources proved to be a robust solution, that has quickly been implemented in businesses around the world.

In my project I gather the data from various pages on the card market website, then I process and save it in a *.csv* file format. These files are then read and converted to dataframes, which are converted to tables in a database. In order for another process to use the data efficiently, it is transformed into various data marts inside the database, creating a data pipeline from the source to the final program, which performs analysis. This way I'm utilizing data warehousing concepts, although I am only collecting data from a single source (one process responsible for data gathering) and delivering it to a single application.

## 2.5 Justification of the thesis topic

How do I use existing technology to solve an existing problem? What questions do I aim to answer?





# Chapter 3

## Specification

What is the actual example that I'm using? Describe the specification from the outside (helicopter view).

What is the result? Peek. Add the server side described.

Implementation of the thesis topic will by nature involve a containerization software (*here Docker*), which will host several subapplications written as semi-standalone scripts, each responsible for a part of the data pipeline. Using Python and its vast collection of libraries I'm handling the data scraping, initial cleaning and maintenance of the pre-stage database in compressed CSV files; as well as managing the MySQL database, creating helper tables, extracting new information and visualizing the data to answer user's questions. With JavaScript, SQL and HTML, I'm able to present the results in form of a simple web application, querying the database connected to a *Node.js* server.

### 3.1 Functional requirements

The system is required to do the following:

1. The system should gather cards, sellers and offers' data and keep it up to date
2. The system should allow the user to choose an expansion of cards to gather
3. The data gathering system should run continuously and gather data once a day
4. The gathering system should visit all card sites from specified expansion and

save (a) card information, dynamic and static, and (b) full information about sale offers of this card to CSV files

5. The gathering system should visit profile pages of all newly-encountered sellers and save their public data to a CSV file
6. The gathering system should keep track of the date and save the date data into a CSV file
7. The database manager system should run continuously and update the database whenever new complete batch of data has been gathered
8. The data miner system should run continuously and create data marts every hour, given the database has the newest data
9. The web application should run continuously, recovering from errors and providing the user with at least four ways of getting useful information from the data

## 3.2 Non-functional requirements

1. The total time of data processing should be less than 6 hours
2. The system should be standalone (bootstrapping itself from *docker-compose.yml* and code), and cross-platform compatible
3. The gathering system should adapt the requests frequency to the server condition
4. No user input is required after running the system

## 3.3 Problem domain

The main goal of this project is to optimize user's card buying decisions basing on the data gathered in the warehouse.

### 3.3.1 Trading card games

Trading card games (*TCG*), also known as collectible card games (*CCG*), are types of card games combining the elements of strategic gameplay and features of trading cards. The first TCG was released in 1993 under the name Magic: The Gathering (*MTG*). The game, released by an eight-person company, was an overnight success, with over 10 million cards sold in just 6 weeks. Two years later, this basement business became a gaming corporation. Today, *MTG* is among the most popular TCGs with roughly 35 million players as of December 2018 [2].

During the game, the goal of each player is to reduce the opponent's life points by strategically playing cards from the hand, before the other one succeeds. However, each player can compile their own deck of 60<sup>1</sup> cards out of the thousands available. This makes every gameplay unique on a level which is fundamentally different from the classic cards. One doesn't have to be an expert in the field to understand that the more cards one has, especially good cards, the higher the chances of winning. Thus, trading the cards becomes an aspect as crucial as the strategy itself.

One of the places where *MTG* cards can be bought is [www.cardsmarket.com/](http://www.cardsmarket.com/) — an online market with a myriad of cards from the game listed for sale, by users from around the world, majority of which lives in Europe. Users are divided into three categories: *Amateur*, *Professional* and *Powerseller*, depending on their setup and *modus operandi* (individuals, zealous hobbyists, card stores). To spend the least amount of money while collecting the most wanted cards, i.e. to optimize the shopping, one would have to analyze thousands of bits of data, from the average prices of all cards, to the velocity of sales and of restocking the virtual shelves.

### 3.3.2 Market-related entities

Each card has a name, an expansion, which is a higher-level grouping of cards, and rarity. These elements do not vary with time. However, the number of cards for sale, and automatically calculated price statistics change every day, that's why these are stored in a separate table, with new rows every day. Similarly, the seller has a name, country of residence and optionally an address, year of registration and what type of seller they are (private vs professional); while the table with sale offers stores every card offer from every seller, from each day. To add a unified time dimension to the

---

<sup>1</sup>Some variations require a deck of 40 cards, which are selected from a random pool of cards

data, a date table is created with unique *date\_id* designating the day, month and year of the datapoint collection.

When the data from the card market is collected, it is stored in five CSV files and one text file. The text file, named after set expansion, contains card names in the order of first visit and it's used to maintain consistent card-to-card progression between runs. Then, each CSV file represents one entity: *card*, *seller* (static entities), *card\_stats*, *date* and *sale\_offer* (dynamic entities).

### 3.3.3 Problem formulation

Questions here.

## 3.4 Data warehouse modelling

The data warehouse lifecycle beings when the data is collected. The pre-stage area encompasses what is written in the CSV files after a gathering run. These files correspond to entities from subsection (3.3.2) and to the tables presented below.

### 3.4.1 Base tables

After the data is gathered from the website, and stored in the CSV files, the database manager updates an image-based MySQL server running in Docker, creating the tables corresponding to each entity — and data miner adds new tables as data marts for future use, and performs other analyses and visualizations.

#### Date

Stores a date dimension with one day increments.

- **id:** smallint
- **day:** tinyint
- **month:** tinyint
- **year:** smallint

- weekday: tinyint

## Card

Stores the cards' immutable information.

- **id**: smallint
- name: varchar
- expansion: varchar
- rarity: varchar

## Seller

Stores the info on website users with at least one sale offer.

- **id**: smallint
- name: varchar
- type: varchar
- member\_since: smallint
- country: varchar
- *address*: varchar

## Card stats

Stores the date-dependent statistics regarding cards price and availability.

- **id**: mediumint
- card\_id: smallint
- price\_from: decimal
- monthly\_avg: decimal
- weekly\_avg: decimal

- `daily_avg`: decimal
- `available_items`: mediumint
- `date_id`: smallint

## Sale offer

Stores the data on sale offers from each day, including who is selling, what and for how much, card condition and others.

- `id`: int
- `seller_id`: smallint
- `price`: decimal
- `card_id`: smallint
- `card_condition`: varchar
- `card_language`: varchar
- `is_foiled`: tinyint
- `amount`: smallint
- `date_id`: smallint

### 3.4.2 Helper views

List the new tables + one sentence about them?

## 3.5 Containers overview

The task of the *firefox-webdriver* container is to provide a constant and reliable access to a browser-like solution for an automated page navigation.

The *mysql\_database* container hosts a database with persistent memory, storing the major part of the data warehouse and making it accessible to other containers.

The *data\_gathering* subsystem automatically scrapes and collects the web data in the *data* directory, while *database\_manager* takes this data and passes it to the database. It is important for the *database\_manager* to be slightly delayed with respect to *data\_gathering*, for reasons that are explained in section (4.1.7).

The *data\_miner* and *node\_server* containers are dependent only on the database, however here boot order and system state are not as important; these asynchronous containers perform their job properly when the data is ready and don't cause critical failure when it's not, which is enough for the application to run reliably.

Each container has its own virtual storage, so to perform tasks on a shared set of data, both docker volumes and data mounts are introduced. Additionally, each container exposes one port internally to a port on the host, inside a network encompassing the whole solution.





# Chapter 4

## Project architecture

The described solution is composed of four directories with subprojects:

- data-gathering,
- database-manager,
- data-miner,
- server,

three auto-generating directories with program-related data:

- logs,
- flags,
- data,

and a `docker-compose.yml` file. Each of the four directories has its own `Dockerfile`, and acts as a build context for that container image. Additionally, two containers are pulled from the Docker image repository, the MySQL server (*mysql/mysql-server*) and Selenium standalone Firefox webdriver (*selenium/standalone-firefox*). Figure (4.1) shows the dependencies between containers.

### 4.1 Containers

Mention `docker-compose` here, `docker` software and composing the solution up and down.

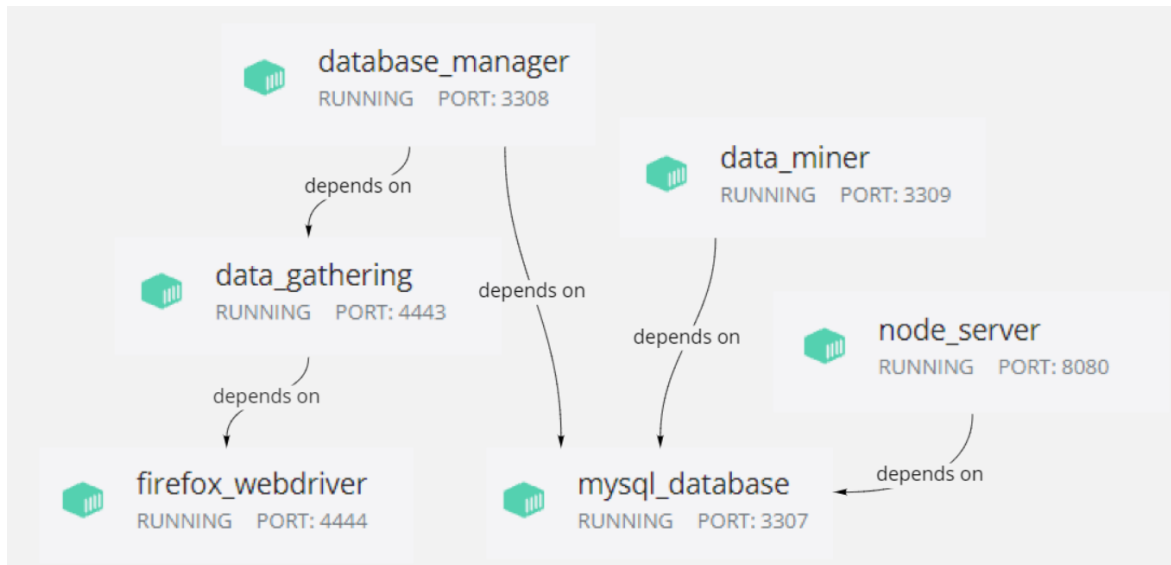


Figure 4.1: Boot order of containers (least dependant first)

#### 4.1.1 Container: *firefox\_webdriver*

This container is to provide a constant and reliable access to a browser-like solution for automated page navigation.

#### 4.1.2 Container: *mysql\_database*

This container hosts a database with persistent memory, storing the major part of the data warehouse and making it accessible to other containers.

#### 4.1.3 Container: *data\_gathering*

This subsystem automatically collects the web-scraped data in the *data* directory

#### 4.1.4 Container: *database\_manager*

This takes this data and passes it to the database. It is important for the *database\_manager* to be slightly delayed with respect to *data\_gathering*

#### 4.1.5 Container: *data\_miner*

These containers are dependent only on the database, however here boot order or system state are not as important; these asynchronous containers perform their job properly when the data is ready and don't cause critical failure when it's not, which is enough for the system to run reliably.

#### 4.1.6 Container: *node\_server*

These containers are dependent only on the database, however here boot order or system state are not as important; these asynchronous containers perform their job properly when the data is ready and don't cause critical failure when it's not, which is enough for the system to run reliably.

#### 4.1.7 Container orchestration

The containers are all started when the `docker-compose up` command is issued in a root directory of the project. This causes the containers from external images to be loaded first, based on the dependency tree. From there, more control over the mutual coordination is available via healthchecks or timing. Out of simplicity, the second approach has been selected for this thesis; i.e. the gathering project is initialized with an immediate 10 seconds delay, so that the webdriver is surely ready before a connection attempt takes place. Similarly, the database managing part is started after 20 seconds, so that the previous container can assess whether there is some new data and any action is needed, or that data can be verified, and marked as ready to be updated to the database. The data mining part is delayed 30 seconds, in case the database is in pre-update state, and other services will start to act on it. However, the most important element of orchestration is shared access to the flags directory, which stores two files with directory checksums, calculated from the ready and complete data after a gathering run. This checksum becomes the new standard and an indicator that the database contents may be outdated.

## 4.2 Services

The three programs written in Python share similar functions. These are grouped into **services** by their main utility and by needed external libraries. Each directory contains a *services* module with the following:

- *logs\_service* — responsible for creating a container-specific logs directory with a daily log file and optionally run log file, general logging to file and console. Libraries needed: **os**, **time**.
- *flags\_service* — this service creates the flags directory, sets up checksum files and provides an interface for everything related to detecting differences in directories or checking the current stage of the data. Libraries needed: **os**, **checksumdir**.
- *data\_service* — handles everything related to CSV files, translating the scraped data into attributes, loading and saving dataframes, pickling and unpickling the data, all data manipulation pre-mining and ensuring the completeness and correctness of the data. Libraries needed: **os**, **sys**, **shutil**, **time**, **pandas**.
- *web\_service* — interface for connecting to the webdriver and navigating the pages, finding appropriate elements on the website, implicit and explicit waiting for response, page validation and so on. Libraries needed: **time**, **bs4**, **selenium**.
- *database\_service* — manages connection to the database, setting up the tables, updating the schema with new data, as well as creating helper views and additional tables in the data mining part. Libraries needed: **mysql.connector**.

## 4.3 Data pipeline

High level overview of the data pipeline (schema). Where does the data come from? How do I get it? Where do I validate it? What happens in the directories? What happens in Docker volumes? What happens in the database? What is the end result of the pipeline? What is the speed of consecutive steps?

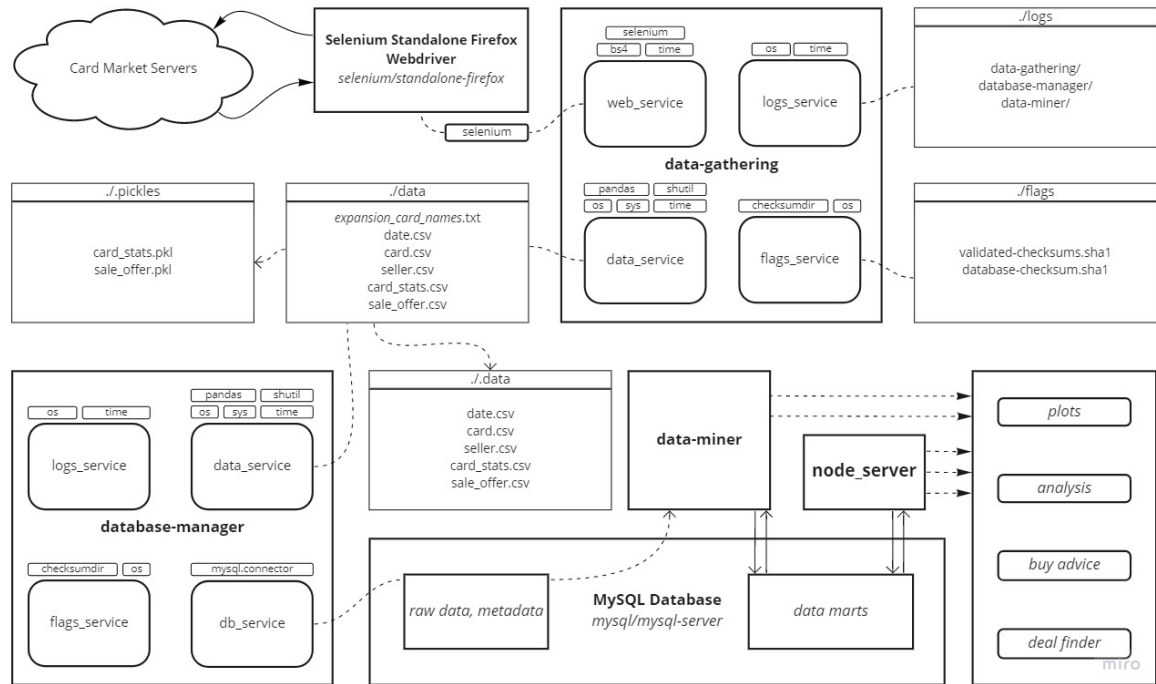


Figure 4.2: Data pipeline from the server (left-top) to results (right-bottom).

## 4.4 Compatibility

Is my project compatible with main operating systems? How does the installation differ on various systems?



# Chapter 5

## Data gathering implementation

What is implemented in this subproject, so that it fulfills its purpose? What does the program do? Step by step.

### 5.1 Program configuration

What is in config.py file? Which values are shared among containers? Which are modifiable and which are not?

### 5.2 Used services

What is my service? (What is a service?) Why do I use modules as services? Why don't I use objective programming? What is the purpose of each of the services in this subproject?

#### 5.2.1 Web service

What is this service all about?

### 5.3 Local directories

How are they setup, what will they contain?

## 5.4 Run scheduling

How is the run scheduled? Does it keep trying when faced with exceptions?

## 5.5 Data pickling and validation

What are the formats of the data? Why is csv compressed? What are pickles for? What were the obstacles (dictionary fiasco)?

## 5.6 Loop over cards

Page url crafting, getting the page, explicit wait. Trying to exhaust the Load more button, explicit wait. Getting the soup object from HTML page source. Decomposition of the soup. Increment START\_FROM. Clean and unpickle the data, revisit scheduling.

## 5.7 Soup decomposition

Getting card name, and adding this card if not present. Card id. Getting card statistics, adding them.

### 5.7.1 Sale offers table

Understanding the table of sale offers. Getting all seller names from the table. Filtering the seller names as a set difference with saved. Iterating over pages of sellers profiles, scraping the data. Sellers added one by one, with `append(seller: dict)` — pickles fiasco. Updating sale offers.

## 5.8 Data pipeline output

What happens when the program is finished? How does the data look, where is it? How do other containers know when to act?



# Chapter 6

## Database management and data mining

Up to this point, our data was stored in the *data* directory and temporary pickle directory. The database managing container takes this data, ensures consistency, and uploads it to the MySQL database.

How is the data stored so far? How it can be stored? Why use database? What kinds of engines we can use? What does the data warehouse contain? What is its purpose?

### 6.1 Program configuration

What is in config.py file? Which values are shared among containers? Which are modifiable and which are not?

### 6.2 Used services

What modules are used in the subproject? What are the differences in the services?

#### 6.2.1 Database service

What does it do, and how does it do it?

## 6.3 Run scheduling

How is the run scheduled? What indicators are present?

## 6.4 Tables from staging area

How do the tables look like when the data is gathered by the first container? What are the data types? What data can be missing or faulty? Do we have any metadata? How can it be used?

## 6.5 Data transformation

How will the data be transformed? What new tables will be created? How will these tables be utilized?

What information is ready to be read right away? What questions can be answered immediately? What questions are more complex? What data can be predicted from the current data?

## 6.6 Program configuration

What is in config.py file? Which values are shared among containers? Which are modifiable and which are not?

## 6.7 Used services

What modules are used in the subproject? What are the differences in the services?

### 6.7.1 Mining service

What does it do, and how does it do it?

## 6.8 Run scheduling

How is the run scheduled? What indicators are present?

## 6.9 Feature extraction

What new features can be discovered? <https://www.analyticsvidhya.com/blog/2021/04/guide-for-feature-extraction-techniques/>



# Chapter 7

## Results

What is generated by the project? What questions are answered? How does my program behave under different cases? Is it resilient?

Check out Docker performance monitoring.

### 7.1 Testing

Testing of the proper functioning of each module. What is the performance of the code under normal and abnormal conditions?

### 7.2 Version control

How was the project developed? How were different versions maintained?

### 7.3 Project variation in time

How did the project evolve with time? What challenges did I encounter? How did I solve problems along the way? Which problems remained?



# Chapter 8

## Summary

What was the subject of this thesis? How did the assumptions or constraints influence the project? What did I implement and how did I do it?

### 8.1 Result assessment

What is the end result of the program? Did I manage to answer all of my questions? Are the effect satisfactory?

### 8.2 Scaling and customizability

How can this project be used outside of the case described in this thesis?

### 8.3 Conclusions

Any other closing remarks.





# Bibliography

- [1] Barry A. Devlin and Paul T. Murphy. “An Architecture for a Business and Information System”. In: *IBM Syst. J.* 27 (1988), pp. 60–80.
- [2] Suresh Kotha. “Wizards of the Coast”. Archived from the original (PDF) on September 1, 2006, retrieved August 11, 2013. Oct. 1998. URL: <https://web.archive.org/web/20060901100217/http://faculty.bschool.washington.edu/skotha/website/cases%20pdf/Wizards%20of%20the%20coast%201.4.pdf>.
- [3] Guido Van Rossum. “The History of Python: A Brief Timeline of Python”. Archived from the original on 5 June 2020. Retrieved 5 March 2021. Jan. 2009. URL: <https://python-history.blogspot.com/2009/01/brief-timeline-of-python.html>.



# Appendix A

## Used symbols

$M(i, j)$  — measure between points  $i$  and  $j$ .



Appendix B

Another appendix