ASSIGNMENT OF BACHELOR’S THESIS

Title: Web Application for Finding Stolen Items in Online Bazaars
Student: Antonín Vlček
Supervisor: Ing. Jiří Novák, Ph.D.
Study Programme: Informatics
Study Branch: Web and Software Engineering
Department: Department of Software Engineering
Validity: Until the end of summer semester 2020/21

Instructions

The goal of the thesis is to implement a tool that makes it easier for individuals to find their stolen items online.

1) Study the topic of scraping content from the web and means of making it searchable.
2) Analyze the current community platforms dedicated to the sale of used things (sbazar.cz, cyklobazar.cz, etc.)
3) Design a scraper to collect information about items that are being up for sale on the community platforms. The scraper must be able to execute requests asynchronously; must store the data efficiently; and must be easily extensible so that useful information can be retrieved from a variety of platforms.
4) Implement the scrapper and a web-based front end, which allows users to interact with the collected data. Keep track of newly posted items that match specified criteria.
5) Test the implemented solution.

References

Will be provided by the supervisor.
Bachelor’s thesis

Web Application for Finding Stolen Items in Online Bazaars

Antonín Vlček

Department of Software Engineering

Supervisor: Ing. Jiří Novák, Ph.D.

July 30, 2020
I would like to express my appreciation for my supervisor Ing. Jiří Novák, Ph.D. who motivated me to pursue this topic. His patience and guidance were indispensable in the process of writing this thesis. A thank you also goes to Ing. Ondřej Guth, Ph.D. for reviewing the thesis and providing valuable feedback.
Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on July 30, 2020
Czech Technical University in Prague

Faculty of Information Technology

© 2020 Antonín Vlček. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author’s permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Abstrakt

Tato práce se zabývá problémy souvisejícími s vyhledáváním odcizených předmětů online implementací systému, který využívá techniky scrapingu k agregování předmětů z online bazarů. Toto řešení umožňuje uživateli prohledávat datovou sadu takto získaných předmětů a nastavit si upozornění, aby obdržel zprávu, když je do datasetu přidána nová položka odpovídající jeho vyhledávacím kritériím. Systém se skládá z více služeb implementovaných docker kontejnery. Tyto služby používají ke komunikaci REST API a toto API je také vystaveno pro použití externím službám. Scraping samotný využívá fronty, které jsou zpracovávány asynchronně pomocí konzumentů postavenými nad frameworkem ReactPHP. Datová sada předmětů je uložena ve fulltextovém vyhledávači Elasticsearch, který poskytuje pokročilé možnosti vyhledávání a je snadno škálovatelný.

Klíčová slova scraping, e-fencing, odcizené předměty online, ReactPHP, Elasticsearch, RabbitMQ, docker
Abstract

This thesis addresses problems associated with searching for stolen items online by implementing a system which scrapes online bazaars and aggregates the postings into one dataset. The solution then allows the user to search through the dataset and set up notifications so that they receive a message when a new item matching their search criteria is added. The system consists of multiple services implemented as docker images. These services use REST API to communicate and this API is also exposed for potential external services to use. Scraping makes use of messaging queues which are being processed asynchronously by consumers built on top of the ReactPHP framework. The dataset of items is stored in Elasticsearch which provides advanced search capabilities and is easy to scale.

Keywords: scraping, e-fencing, stolen items online, ReactPHP, Elasticsearch, RabbitMQ, docker
# Contents

## Introduction
- Motivation .............................................. 1
- Goal .................................................. 1
- Structure of the thesis .......................... 2

## Theoretical part
1. E-Fencing – stolen goods being sold online .......... 3
2. Existing solutions ................................... 4
   1.2.1 Aggregators of reports & social media groups ... 4
   1.2.2 Specialized databases ............................ 4
   1.2.3 General search engines ......................... 4
   1.2.4 Manual search through online marketplaces ... 5
3. Web Scraping ......................................... 5

## Analysis and design
1. Domain terminology .................................. 7
2. Functional Requirements ............................. 9
3. Wire-frames .......................................... 10
4. Non-functional Requirements ...................... 10

## Realization
1. Architecture .......................................... 15
2. Banot API ............................................ 17
   3.2.1 REST API ....................................... 17
   3.2.2 Framework ..................................... 20
   3.2.3 Endpoints ...................................... 22
3. Banot SDK ........................................... 30
4. Scraper ............................................... 32
   3.4.1 Process of scraping .......................... 33
<table>
<thead>
<tr>
<th>3.4.2 Scrape List Pages</th>
<th>36</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.3 Scrape Detail Pages</td>
<td>37</td>
</tr>
<tr>
<td>3.5 Web</td>
<td>38</td>
</tr>
<tr>
<td>3.5.1 Website screens</td>
<td>38</td>
</tr>
<tr>
<td>3.5.2 Technology &amp; tools</td>
<td>41</td>
</tr>
</tbody>
</table>

4 Example use case & Testing

| 4.1 Create a Web Resource | 43 |
| 4.2 Run scraping | 44 |
| 4.3 Search through items | 45 |
| 4.4 Create a report and send notifications | 46 |

5 Ideas for further work

| 5.1 Graphical interface for adding new Web Resources | 51 |
| 5.2 Google Cloud Vision | 52 |
| 5.3 Production infrastructure | 52 |

Conclusion

A Acronyms

B Bibliography

B Contents of enclosed CD
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Relationships of Root, List, and Detail Pages</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Wire-frame of the homepage</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Wire-frame of the search page</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Wire-frame of the report page</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Diagram of the architecture</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Architecture of the Apitte framework</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>ERD of WebResources, Instructions, and Root Pages</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Individual endpoints manipulating with Web Resources</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>Individual endpoints manipulating with Items</td>
<td>26</td>
</tr>
<tr>
<td>3.6</td>
<td>Individual endpoints manipulating with Reports</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>Example use of the Banot SDK Client</td>
<td>32</td>
</tr>
<tr>
<td>3.8</td>
<td>Example of List and Detail Page</td>
<td>33</td>
</tr>
<tr>
<td>3.9</td>
<td>Show Report Form</td>
<td>38</td>
</tr>
<tr>
<td>3.10</td>
<td>Item Card Control</td>
<td>39</td>
</tr>
<tr>
<td>3.11</td>
<td>Examples of Lucene queries</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>Definition of the sbazar.cz as a Web Resource</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>The search bar</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Confirmation message sent when report gets created</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>Notification message sent when new matches are found</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>DataflowKit “Build your Scraper” interface</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>The Detect Labels functionality of Google Cloud Vision service</td>
<td>53</td>
</tr>
</tbody>
</table>
Introduction

Motivation

I bought myself a nice road bike to do my daily commute and some sport on the side. It did not take long before I found myself picking up the remains of my wire lock from where my bike had been parked just an hour ago. Upon arrival, the police officers did not give me much hope and recommended I “conduct my own search” and to look for it online.

As it turns out, there are many online marketplaces to look through with many new items being posted every day. In some cases the marketplace allows the user to sign up for email notifications when a new item is posted, but this would not help if my bike had already been posted, also even this functionality on its own is rare and on many sites a manual search is the best one can do.

It might be too late for my bike, however there are more bikes being stolen every day. Also bikes are not the only article of interest: laptops are being stolen from libraries and coffee places, houses are being robbed and there is a good chance that at least some of these items do end up in an online marketplace one way or another.

Goal

The goal of the thesis is to implement a tool that makes it easier for individuals to find their stolen items online. To achieve this the solution needs to aggregate items posted in online bazaars, allows users to submit reports of stolen items which are then periodically evaluated against the available data, and notify the user when matches are found. To stay relevant the system needs to make it easy for new online marketplaces to be added or adjusted if the layout of an existing website changes. To operate efficiently the system needs to allow
Introduction

multiple pages to be scraped at the same time.

Structure of the thesis

To start with this thesis outlines the challenges associated with finding a stolen item online, and it is setting a goal of building a technology to improve the current state of things. Subsequently, in the Theoretical part, it describes the practice of, so called, E-Fencing – stolen goods being sold online to show how and why do stolen items end up in online marketplaces. This is then followed by a research of existing solutions and a brief description of the approach taken in this thesis.

The Analysis and design chapter describes a bird’s eye view of the technical solution and defines the domain terminology. Having gone through this background, the Realization chapter drills deeper into the specific components which make up the solution and provides detailed technical information.

Finally, the Example use case & Testing chapter puts all this into practice and presents a real-world example scraping and then searching through items from few online marketplaces. To conclude the author lists out ideas on what improvements could be made in the future and where there is a room for further research.
This chapter presents research into the topic of E-Fencing. It is important to analyze this phenomenon because it is the reason why there is the need to search for stolen items online in the first place. This is then followed by an analysis of the currently available solutions that people turn to in the effort to recover their stolen items and in the hope of finding them somewhere on the Internet. Finally, this chapter describes the technique of Web Scraping which will be instrumental for the solution that this thesis develops.

1.1 E-Fencing – stolen goods being sold online

Theft is a phenomenon which has been around since the dawn of civilization and it will most certainly be with us for the foreseeable future. Once a thief takes hold of an item they have to find a way to dispose of the item and make profit. This often involves a fence – a middleman – who buys the item for cheap and then sells it in a more legitimate market with profit. The mainstream use of the Internet with its plentiful e-commerce sites, online auctions, classified adds platforms, and other means of selling goods opens up a new pathway for fences or even the thieves to offer their items directly to the end consumer. The practice of selling stolen goods online is called “E-Fencing” or “eSelling”. The anonymity of the Internet, the quick turn around, and the vast amounts of items being posted on daily basis, are just a few factors which lower the perceived risk by the offenders while increasing they reward when compared to offline methods. Regardless of these factors the existing academic literature is still catching up and does not contain much information on this topic.

An explorative study from 2018 analyzed cases of theft where the item was disposed of using the Internet as a platform. These cases were categorized based on the type of the website (classified adds site, auction sales sites, and
1. **Theoretical part**

5 other categories) and the type of the stolen goods. This analysis showed that the most common were items in the category of *Bikes, motorcycle, cars or trucks and parts* sold through classified adds sites. The examples provided later on in this thesis revolve around stolen bikes, however the solution can be applied to other types of items as easily.

A different paper [2] is applying artificial intelligence techniques to categorize the items sold online and attempts to identify patterns which make the items “suspicious” and more likely not to have a legitimate origin.

From the information presented in this section it is clear that E-Fencing is an important issue and a one where more research is still needed.

1.2 **Existing solutions**

This section analyzes a few solutions that victims can turn to when searching for stolen items regardless of whether these items were posted online or not. From groups on social media, through specialized database to general purpose search engines, there is variety of options.

1.2.1 **Aggregators of reports & social media groups**

Some specialized websites provide a section where people can post ads of their stolen items. An example of this can be the Czech website cyklobazar.cz which has a section call “stolen bikes”. [4] There are also numerous groups on social media where people post reports of stolen items. These solutions might help in case that the potential buyer of this stolen good would actively search to verify that it has not been stolen. Also thanks to these solutions there is a chance that someone who sees the posting will notify the victim.

1.2.2 **Specialized databases**

Staying in the bike category, there are also solutions which allow owners to enter their bike frame numbers into a database which can be checked when a suspicious item is found. These increase the chance of the bike being recovered and theoretically give police the means for checking the items. [5][6]

1.2.3 **General search engines**

As a victim searching for a stolen item online, people might also turn to search engines such as Google to try to find their items. While this might help, there are a few things which make this search inefficient. Generic search engine algorithms (such as Google’s PageRank) are optimized for horizontal rather than vertical searches [7] and have taken up the task to index everything on the public internet which takes a toll on how quickly a new content can be
1.3. Web Scraping

indexed. It is hard to say exactly how long it takes before Google indexes new content since there are many factors, according to it can take something between 4 days and 4 weeks, which is inadequate when searching through quickly changing databases of online bazaars.

1.2.4 Manual search through online marketplaces

At the end of the day many people will end up manually searching through the different online marketplaces. Rarely, these platforms allow users to specify filters and notify them about newly posted items which match these filters. Even when the notification functionality is available it only triggers for newly posted items and it does not work retrospectively. On top of this, there is a great variety of online auction sites, classified ads sites, and other platforms which makes it nearly impossible to keep up with newly posted items.

The intention of this section was not to condemn the existing solutions. These solutions might still be helpful and complementary to the solution proposed in this thesis. This overview points out shortcomings and problems so that they can be addressed when designing and developing the system in this thesis.

1.3 Web Scraping

As it has been mentioned at the top of this theses one of the main goals is to aggregate items form online bazaars and Web Scraping is the technique which can help achieve this objective. It is defined as “using technology tools for automatic extraction and organization of data from the Web for the purpose of further analysis of this data.”

Even though Web Scraping as a method of data extraction and organization is as nearly as old as the World Wide Web itself, the first web robot - World Wide Web Wanderer - was created in June 1993, there is still a great deal of controversy surrounding the legality and ethics of this practice. According to and there are few underlining legal principles which are applicable when dealing with cases related to Web Scraping but there still is no legislation for Web Scraping specifically.

A research paper from 2018 proposes a set of 7 questions which should be considered for each project dealing with Web Scraping to make it legal and ethical:

1. Is web crawling or web scraping explicitly prohibited by the website’s “terms of use” policy?
2. Is the website’s data explicitly copyrighted?
1. **Theoretical part**

3. Does the project involve illegal or fraudulent use of the data?

4. Can crawling and scraping potentially cause material damage to the website or web server hosting the website?

5. Can the data obtained from the website compromise individual privacy?

6. Can the data obtained from the website reveal confidential information about operations of the organizations providing data or the company owning the website?

7. Can the project requiring the Web data potentially diminish the value of the service provided by the website?
This presents the name of the system that will be built before moving onto what needs to be in place for the system to fulfill the goal which was set at the start of this thesis. It lists the functional requirements, which provide a scope for the functionality that needs to be implemented in the final solution. Subsequently it identifies the domain specific terminology so that these terms can be used throughout the thesis and the systems codebase. Later in this chapter an analysis of the user interface is provided by listing wire-frames of the key pages. Finally, before handing over to the next chapter on Realization, the non-functional requirements are specified. This analysis then drives the architecture and design decisions.

Since the main benefit that this system will provide to the end users is the ability to send Bazaar NOTifications the system was named Banot. Names of some of the services which are described in section 3 use this name so that the name of the service is distinguishable from generic terms.

2.1 Domain terminology

This section analyzes and codifies the high level terminology which is then used throughout the thesis and the codebase of the system. Describing these terms and their meaning at the beginning is essential since the chapters that follow rely on these terms to describe other concepts within this domain.

**Web Resource** - There are many different terms that could be used to describe the portals from which the items are being scraped. Indeed, terms like online bazaar, online marketplace, or classified adds site all describe the potential source of information for this system. Since all of these can be considered as resources from which the system extracts information and all of these resources are located on the World Wide Web, the term Web Resource
will be used throughout the thesis and the codebase.

**Item** - This term can refer to the physical object that it being sold, the posting or the add that is being published on the Web Resource, or it can refer to the data structure which contains the data extracted from the Web Resources about a posting. Throughout the thesis this one term is used to refer to all of these, however from context the specific meaning is always clear. In the codebase this term refers exclusively to the data structure which holds the extracted information.

**Instruction** - When processing the content of the pages downloaded from Web Resources the system will need to be instructed on how to locate and extract specific information about the items. An Instruction, sometimes also referred to as the parsing instructions, provides this information.

**Detail Page** - The web page which presents all the (publicly) available information about an Item is refereed to as the Detail Page throughout this thesis.

**List Page** - All the analyzed Web Resources organize its content in a way where there are pages with list collections of items, these can be the landing pages for specific categories of items or search result pages. The item elements listed on these pages then contain a link to the Detail Pages. These pages are also typically paginated which means that they contain a link to one or multiple other pages that also list items. Such page is referred to as the List Page in this thesis.

**Root Page** - In simple terms, the Root Page is the first of a given set of List Pages. Because the term List Page refers only to an individual page form the paginated set, the term Root Page is needed. From this perspective the Root Page also identifies this collection of List Pages. When taken one step further the Root Page also identifies a collection of Items because each List Page does as well. Each Web Resource consists of multiple Root Pages. Figure 2.1 provides a visualization of this relationship between Root, List, and Detail pages.

**Report** - The Report represents the user’s intend to be notified when an item matching specified criteria is added to the dataset. To create a Report the user needs to specify the criteria which make an item a match, provide an email address to which the notification will be sent, and choose how often they want to be notified.
2.2 Functional Requirements

This section lists seven functional requirements. The first three are aimed at the scraper itself while the rest focuses on the interface facing the end-user. These requirements provide the scope for what needs to implemented as part of this thesis.

**FR 1 Scrape various online bazaars** - The scraper can extract data from multiple online portals. These resources can be added without the need to change the source code.

**FR 2 Variable parameters of an item** - Information collected about each item can differ depending on the platform.

**FR 3 Periodical scraping** - The dataset of items can be kept up to date by running the scraping periodically and updating the existing items.

**FR 4 User can search** - The user can search through the dataset of items scraped from the various online bazaars. Each result provides information about the item and allows the user to open the original location of the item on the web.

**FR 5 User can create a report** - The user can create a report of their stolen item to set up notifications so that they receive a message when new items are added into the dataset.
FR 6 **User is sent a notification** - When a new item matching the search criteria of a report is added to the dataset a message is sent to the user who created this report. The message lists some of the newly found items and links the user to the full list.

FR 7 **User can view their report** - User can view their report and browse the collection of items which are identified as matches based on the search criteria specified by the user.

### 2.3 Wire-frames

With the functional requirements specified wire-frames are a good tool to visualize how these requirements affect the user interface. What the wire-frames cannot capture are all the complexities of scraping and data manipulations that happen in the background unbeknownst to the end-user. Yet they can still have some effect on the decisions made in those areas as well. This section consists of three figures presenting three different user facing pages. The wire-frames are just drafts and should provide context rather than define the final UI, examples of the built UI are presented in chapters Realization and Example use case & Testing.

Figure 2.2 shows a wire-frame of the homepage. This page should direct the client towards search for an item, allow them to open up an existing report and provide some context about the system as a whole and how does it work. It could also list some interesting statistics.

The next wire-frame from figure 2.3 presents the search page. The focus here should be on a very simple UI which makes it easy for the user to define their search query, browse through the results and create a new report for the defined search criteria. Each item element should show the available information about that item including the title, description, and a photo. Each item element should also provide a link to the original page hosted on the online marketplace platform.

Finally the page where the user can view and manage their report is shown in figure 2.4. The same item elements as on the search page can be used here as well. The user should have an option to pause the notification or delete the report all together.

### 2.4 Non-functional Requirements

Contrary to the functional requirements from section 2.2 these do not focus on what functionality needs to be included in the system but rather specify how the system should operate internally. Section 3.1 builds on top of these
2.4. Non-functional Requirements

Figure 2.2: Wire-frame of the homepage.

requirements to establish the architecture and the technologies for implementation of this system.

NFR 1 **Scrape asynchronously** - During scraping a large amount of time is spent waiting for the content to be retrieved from the host, with a synchronous approach all this time would be lost due to the execution being blocked on that one request. For the reason the scraper must be able to operate asynchronously and be able to process multiple requests at the same time.

NFR 2 **Store items in NoSQL database** - To persist the potentially very large amounts of semi-structured data of the items, these entities should be stored in a NoSQL database.

NFR 3 **Run in containers** - The services that this system will consist of need to run in containers so that when the system is deployed the development and production environments are as similar as possible.

1Concurrency is however not required.
2. Analysis and design

Figure 2.3: Wire-frame of the search page.

**NFR 4 Multiple instances of the Scraper** - The service responsible for scraping needs to be able to run in multiple instances so that it is scalable.

**NFR 5 REST API** - The internal services should use REST API among themselves. The whole system should provide REST API which can be used by external services as well.
Figure 2.4: Wire-frame of the report page
Chapter 3

Realization

This chapter focuses on the individual services that from the system, it is outlining the technologies used to implement them and it comments on technical challenges encountered along the way. It starts by presenting the architecture of the system before moving onto describing the central API component. A section is also dedicated to the SDK built to simplify the implementation of this API into the other two services. One of them is the Web service which provides a user friendly interface to interact with the system. And the other is the “workhorse” of the system the Scraper service which is responsible for extracting the data from the online bazaars.

3.1 Architecture

As it can be seen in the diagram 3.1 there are multiple services that form the whole system. Each of these services performs its specific tasks. The following sections will describe the responsibilities of these services in more detail as well as drill down into the decisions made during implementation.

There are three main services which contain most of the logic of the system:

- The public facing website provides a user friendly way to search through the dataset of scraped items and to set up notifications which inform the user about newly added items.
- Banot API provides APIs for Resources, Reports, and Items. Manages when pages are scraped by pushing set of root pages into the List Page Queue.
- Scraper is a consumer of the List Page Queue, it extras Detail Page URLs from the List Page and scrapes the information from the Item
3. Realization

Figure 3.1: Diagram of the architecture.

Detail page. It then sends a request to the Item API of the Banot API to store the Item. There can be many instances of this service running at the same time.

Apart from these main services there are two storage services:

- **MySQL database** stores the relational data, the Resources, Root Pages, Instructions, Reports, and Matches.
- **Elasticsearch database** stores the non relational Item data.

*RabbitMQ* is used to control the flow and maintain a queue of List Pages which need to be scraped. The queue of details items which need to be processed is held in memory of each scraper instance.
Finally the system requires a Scheduler which is responsible for initiating periodic tasks such as scraping and notification sending.

Initially there was an intent to have three separate services instead of the Banot API. Each of these services - Reports API, Resources API, and Items API would expose its own API and would abstract the other services away from it’s inner workings in the spirits of the microservice architecture. However during implementation it turned out that this creates an unnecessary overhead in terms of the deployment a development environments. For that reason the idea was dropped and the services were consolidated into one Banot API. The service which would benefit the most from being separated out would be the Item API, due to its frequent interaction with the Scraper instances it has very different requirements when compared to the Reports and Resources Modules.

3.2 Banot API

All of the services from figure 3.1 are essential for the system to perform its function, without the Scraper Service the system would not be able to capture and parse the data, without the Web Service the end-client would not be able to search through the data in a user friendly way. However the Banot API can very well be considered as the central piece of the system. It standardizes the way the other services talk to each other, it provides a simple interface for “the outside world” to communicate with the whole system, and finally it performs actions which do not fit into the scope of any other single service.

This section describes the responsibilities, design decisions and technologies used to implement Banot API. The next few paragraphs briefly describe REST API, an architectural style which is essential for this service. Best practices and the ways in which Banot API follows them is also discussed before moving onto the framework that is used at the core of this service. Finally this section dissects the main three groups of endpoints - the Items, the Web Resources, and the Reports.

3.2.1 REST API

To fulfill its role of providing an interface to the system this service uses the architectural style of REST API. Use of this style has been gaining popularity for a long time [12], REST stands for Representational state transfer and it is a software architectural style that defines a set of constraints to be used for creating Web services. Web services that conform to the REST architectural style, called RESTful Web services, provide interoperability between computer systems on the Internet. RESTful Web services allow the requesting systems
“Web resources” were first defined on the World Wide Web as documents or files identified by their URLs. However, today they have a much more generic and abstract definition that encompasses every thing, entity, or action that can be identified, named, addressed, handled, or performed, in any way whatsoever, on the Web. In a RESTful Web service, requests made to a resource’s URI will elicit a response with a payload formatted in HTML, XML, JSON, or some other format. The response can confirm that some alteration has been made to the resource state, and the response can provide hypertext links to other related resources. When HTTP is used, as is most common, the operations (HTTP methods) available are GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS and TRACE.

By using a stateless protocol and standard operations, RESTful systems aim for fast performance, reliability, and the ability to grow by reusing components that can be managed and updated without affecting the system as a whole, even while it is running.

The REST architectural style is defined fairly loosely, and so there are numerous best practices, which, when followed, make the API easier to maintain and use. What follows is a list of these practices that Banot API uses.

**Versioning**

Once an API is exposed, either publicly to the Internet or even internally to other services in a system, it can start being consumed by these 3rd party clients. These clients rely on the structure of the exposed endpoints and the data passed as responses. It would be naive to assume that the first version of the API is the final one, the domain requirements of a problem will inevitably change for one reason or another and a backwards incompatible changes will need to be introduced. To prevent breaking the clients downstream the API server should version its endpoints. The introduction of a backwards incompatibility then results in an introduction of a new version. There are different ways to communicate the version, Banot API requires the clients to specify the version in the URL address of every endpoint that they want to call. For example, the endpoint used to search through the dataset of items would be called as follows:

```
GET {baseUrl}/v1/items?query=[query]
```
3.2. Banot API

Authentication

Now that the client and the server have a way of communicating the version of the API, the client can query the right endpoints. However just knowing what you want does not mean you will always get it. The client first needs to authenticate itself, only when successful the server will send over the data, otherwise the 401 Unauthorized status code is returned. There are a few different authentication methods and strategies that can be used. Banot API is using the api-key method to authenticate its clients. The API key is currently being stored as a secret in the environment of the server. While that is a secure approach this might soon need to be improved by storing multiple api keys in a database. Internally the authentication is implemented using the middleware pattern. Each request passes through the authentication middleware which determines whether to continue and pass the request further into the application or whether to directly return a response informing the client that it is Unauthorized. The middleware pattern is just one of the design patterns put in place by the framework which is used to built the Banot API. The framework is described in more detail in the section.

Documentation - OpenAPI

Using REST API with versioning and authentication removes some of obstacles for computer systems to talk to each other, but there still remains the problem of how to show people, to the developers of the clients who will consume the Banot API, how to use the API and explain the structure of the data that is being returned. This is where documentation comes into play. As with the other aspects mentioned above there are different approaches available. One way to simplify the use of an API is to build an SDK, a library which the developer can require as a dependency of the client application. Using this library abstracts them away from the API itself. Indeed, the section outlines an SDK built for the Banot API. Even though the SDK can be big help to developers, it is by no means the silver bullet solving all of the problems around documentation. For one, an SDK is language specific, thus it needs to be written and maintained separately for each language. These days there are tools which can generate and update an SDK automatically, but even still one of the main benefits of the REST architectural style is its platform independence and only relying on SDKs would inevitably lead to limitations in terms of the technology that the client can use.

On the other side of the spectrum entirely, the documentation could, in theory, take the form of an article which describes the endpoints in English sentences with some tables and diagrams describing the structure of the responses. It is easy to see how that would be hard to produce and maintain. It would most

\footnote{Swagger Codegen or Apimatic are just some examples of such tools}
likely be also very hard for users to navigate. Also the documentation for each API would be entirely different which would further steepen the learning curve. There is a need for a solution somewhere in the middle, something that is structured, standardized, easy to produce, is capable of providing examples and user friendly description where needed. As it turns out this is exactly what OpenAPI (formally swagger) was built for. The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection.\[19\] Banot API is making use of this OpenAPI standard by annotating its endpoints, their parameters, request and response objects. The framework that powers the Banot API, makes it possible to expose an endpoint which returns the OpenAPI Specification document with information that is always up to date with the current API. This JSON document can then in turn be loaded into a service which turns this into an interactive documentation.

** * * * **

Banot API is built as a RESTful service which makes it well suited as the central piece of the system and an interface through which both the internal and external services can communicate. The API is versioned using the version as part of the URL of all endpoints which makes it possible for new functionality to added without braking backwards compatibility. Clients need to authenticate using the api-key method. Finally, there is also a documentation in the form of the OpenAPI Specification which is automatically kept up to date. Apart from the documentation there is also an SDK which is described in more detail in section 3.2.

### 3.2.2 Framework

This section introduces the technology used in the implementation of Banot API. The main focus is on the framework that powers this service. This section briefly describes the architecture of the framework and comments on the usage of these concepts in the implementation.

Banot API is built in PHP. It uses version 7.4 which is the most recent stable version at the time of writing. Rather than building everything from scratch a framework is used. The framework chosen for this task is called Apitte, it is a modern API framework build on top of Nette Framework.\[20\] The framework consists of multiple packages so that each application can chose to use only what it really needs. Banot API makes use of the core, debug, middlewares,
3.2. Banot API

and *openapi* packages. These packages provide some essential functionality such as routing of the requests to the correct method in a controller class, processing and validation of request parameters, logging errors, OpenApi documentation and more. Figure 3.2 shows the architecture of this framework and the next few paragraphs provide a brief description of the core concepts according to [20].

![Architecture of the Apitte framework](image)

**Request and response** objects both implement the PSR-7 interface which is an industry standard for mapping HTTP requests and responses onto objects in PHP.

**Middlewares** are optional. When used they transform and validate request or early return response before it is handled by a dispatcher. In Banot API a middleware is used for authentication.

**Dispatcher** is a front controller of whole API. Its responsibility is matching request via a router, invoking endpoint through a handler and returning response from that endpoint. It also sends 404 response if a request is not matched with any endpoint.

**Router** checks if an endpoint from a schema matches a request. Internally all annotations from controllers are used to build *api schema*. The schema is basically metadata describing the whole API. Endpoints, routing, and OpenApi.

**Decorators** are also optional and they are used for transformations of a request before it is passed into an endpoint and for transformations of a response after it is returned from an endpoint.

**Endpoint** is a representation of a unique URL (for instance */v1/items*) and one or multiple operations (HTTP methods). An endpoint is implemented as a method of a controller class.

Banot API adjusts the framework in few places to fit its needs. Most importantly, there is a custom dispatcher used so that DTOs can be easily serialized to JSON using the *symfony/serializer* [21] library.

The DTOs themselves can be broken down into two categories: requests and
3. Realization

responses. The Request DTOs are used to normalize and validate the body of an incoming request. This makes it significantly easier to work with the data later on in the application, while it also decreases the logic needed directly in the controller methods. The Response DTOs are used to standardize the content of a response. If for example details about an item would need to be returned from multiple endpoints, all of these endpoints could make use of the same Response DTO. And yet again, having the responses wrapped in an object makes it possible to separate some of the logic related to serialization outside of the individual controller methods which in turn makes the controller methods shorter and easier to understand.

There are also few “rules of thumb” which are not specified or enforced by the framework itself, however these rules make it easier to separate the business logic into the appropriate classes and to keep the codebase more manageable:

- Controller method is only responsible for: Processing the parameters and the body of a request. Calling the appropriate method in a facade. And finally, formatting the success output, oftentimes using the Response DTOs, or catching exceptions and reporting errors to the end-user.

- Each controller should contain only one method.

- A facade cannot depend on another facade. This prevents circular dependencies and promotes clear separation of responsibilities.

* * *

Banot API is implemented in PHP 7.4, it builds on top of the Apitte framework while adjusting the functionality of the framework by providing its own dispatcher. When processing requests and responses an effort is made to limit the work with plain arrays as much as possible and so Request and Response DTOs are used. Last but not least, there are rules designed to help to keep the codebase manageable.

3.2.3 Endpoints

With the fundamentals of the architecture explained, this section shifts the focus onto the functional aspects of the service looking at its different endpoints. The endpoints can be loosely categories into 3 groups based on which entity

\[\text{4}\text{Most of this is done using annotations and the Requests DTOs}\]

\[\text{5}\text{In a situation where multiple facades rely on the same logic, the logic in question should be separated into a service which is then injected into all of the facades.}\]
they deal with. For each of these groups it describes the implementation of technologies used to perform the operation at hand.

**Web Resources**

One of the main objectives of the whole system is to scrape items from online bazaars and make them searchable from one place. Before it is possible to start scraping the details of particular items, a database of Web Resources, the entities representing the online bazaar portals, is needed. Endpoints in this group cover the essential functionality of creating, retrieving and deleting of the Web Resources and other closely related entities. There are also two endpoints which initiate the scraping of Web Resources. Individual endpoints are listed in figure 3.4.

![ERD of WebResources, Instructions, and Root Pages](image)

**Figure 3.3: ERD of WebResources, Instructions, and Root Pages**

As the ERD diagram from figure 3.3 shows, there are two other entities related to the main Web Resource entity. Both Instruction and Root Page are important for scraping and their significance is explained in more detail in section 3.4. In this section it just important to know that each Web Resource is associated with a set of Instructions and Root Pages. Instructions are used to parse the content of a page downloaded from the Web Resource and Root Pages represent the starting point of scraping - typically this would be a landing page of a category listing the first few items from a larger paginated set.

Because of the relational nature of this data, MySQL database was chosen to persist this information. Within the codebase the technique of Object Relation
3. Realization

Figure 3.4: Individual endpoints manipulating with Web Resources

GET /v1/resources/?from=[int]&size=[int]
GET /v1/resources/[resource-name]
DELETE /v1/resources/[resource-name]
POST /v1/resources
POST /v1/resources/scrape
POST /v1/resources/scrape/[resource-name]

Mapping (ORM) is used to manipulate the entities. The ORM is build on top of NextrasORM framework. Within this framework there are three distinct layers:

- The *Entities* layer uses annotations to describe the structure of the data, entities are created either by the user or during a process called hydration when the raw data retrieved from the database is used to populate the entities with that information.

- The *Mapper* layer is in place to abstract away any differences between different database systems, since NextrasORM also supports Postgres and MS SQL Server alongside MySQL.

- The *Repository* layer provides an interface for retrieving, persisting and removing of the entities.

Apart from providing an interface for the manipulation of these three entities, there are also endpoints used for scraping. As it is described later in section 3.4 the scraper uses messaging queues, specifically RabbitMQ, to continually process the large number of pages that need to be scraped. Once there are some pages in the queue, instances of the Scaping Service operate as both producers and consumers of the queue and keep the process going until all pages that need to be scraped are downloaded and parsed. However there is still the need for someone to push the first pages into the queue. This is exactly what the scraping endpoints do. By querying the database the Root Pages associated with one or all Web Resources are retrieved and then pushed into the messaging queue which kick-starts the process of scraping. In a production environment these endpoints are called periodically so that the dataset of items is kept up to date and users can be notified of newly posted items that match their queries.
Finally, having these endpoints exposed to the internet allows other authorized services to contribute content to the system. This is important primarily because one of the functional requirements states, that the Web Resources have to be easily added. Having all the resource-specific information stored in a database and being able to download and parse pages just using this information goes a long way since code changes are not needed to add new bazaar portal, however it is the exposed API which makes it possible for 3rd parties to add new Web Resources. This means, that even though there currently is not a way to add or delete a resource from the Web Service (the front-end of this system described in more detail in section 3.5), it can be either easily extend or a new service entirely can be built for this purpose. Section 5.1 proposes a graphical user interface which could be built to allow non-technical users to add Web Resources with parsing instructions on their own in a visual, user friendly way.

Items

Given a dataset of Web Resources with Instructions on how to retrieve the content of the Detail Pages and how to extract specific values (such as the title or a price of an item), it is important to look at how are these values stored and how is this data used throughout the system. On one side, the values are being parsed out of raw HTML content within the Scraper Service, however the Scraper is not responsible for storing these values, it passes this responsibility onto the Banot API. On the other side, there is the Web Service which needs to execute searches on behalf of the user to provide them with relevant results. This section first describes the implementation of Elasticsearch, the storage and search engine used for the scraped items, before moving onto the specific endpoints which make it possible for the other services to create, update, and search through the scraped items.

Elasticsearch - As already outlined in section 2.4 the Items do not need to have a relational structure, the fields that need to be stored for each Item may differ, finally the storage needs to be optimized for large volumes, scalability, and full text searching. Combining this criteria led to the choice of Elasticsearch as the technology to power the persistence and search of Items.

When implementing Elasticsearch in a PHP application there are two popular libraries elastica and elasticsearch-php. A comparison of these two libraries can be found in [22]. The main difference being that elasticsearch-php provides a very low level approach where clients accept simple associative arrays as parameters. All parameters, from the URI to the document body, are defined in the associative array. [23] This in combination with elasticsearch-php being more popular over all [22] led to the decision to use that library in Banot API.
3. **Realization**

To make the codebase more maintainable and to make it easier to work with the responses from elasticsearch. A new namespace, `App\Search`, is introduced into the application. This namespace contains:

- **DTOs** that wrap the responses from elastic search. This makes it easier to handle errors and process the documents returned from elasticsearch.

- **Document entities** which represent the documents stored in elasticsearch. Currently there is only the `Item` document.

- **Collection class** which is when a set of Documents needs to be passed around the application.

- **Search service** is essentially a wrapper around the elasticsearch-php library which is only called from this service.

- **ItemsIndex service** which extends the `AbstractIndex` class and centralizes all the operations with the `Item` Document.

**Endpoints** - To satisfy the demands of the Scraper and Web services Banot API exposes three endpoints which add new or update exiting items, search through the existing items, based on a set of URLs check whether a specific items can be skipped during scraping because the system already contains up to date information. Figure 3.5 lists these endpoints.

![Individual endpoints manipulating with Items](image)

**GET** `/v1/items?query=[lucene-string]&from=[int]&size=[int]`

**PUT** `/v1/items`

**POST** `/v1/items/need-scraping`

To make it easy for the Scraper service to persist the data it scrapes, the data is passed into the same endpoint regardless of whether an item is being added or updated. The API validates and saves the data as a document into an Elasticsearch index. In this sense Banot API is being used as a proxy for Elasticsearch which also exposes its own REST API. There are few reasons why this overhead of passing data from Scraper to Banot API and then to Elasticsearch is worth the development effort and the extra time spent during each request. Firstly, having this intermediary step future proofs the whole system, it gives us more control over how to validate and process the data.

---

"The raw associative arrays are very verbose."
before sending it to storage. Secondly, it is decreasing the amount of logic that needs to be implemented in the Scraper service itself. This is a benefit because the Scraper is designed to focus only on downloading and parsing the content of a page, data persistence thus lies outside of its scope. Finally the extra time added by this step should be negligible since all requests in the Scraper service are being executed asynchronously and on top of that all three involved services run on the same cluster when in production environment which makes the communication between them very quick when compared to the queries the Scraper needs to make when downloading content from the Internet.

With Items added, the next endpoint deals with searching. The query string passed in supports full Lucene syntax \[^{24}\] which gives the user a lot of options to narrow down their search criteria. In a very similar “proxy” fashion as described in the previous paragraph, this query is passed into Elasticsearch, the result then hydrated into the Document and Collection classes before being serialized and returned to the client of Banot API.

Finally, the /v1/need-scraping endpoint, accepts a set of Detail Page URLs, and returns back only those which need to be scraped. This is useful since it may save many queries to the Web Resource (the online bazaar) which makes the scraping more efficient and also decreases the load on the host. To determine whether a URL needs to be scraped a query is sent to Elasticsearch asking whether an item with a given URL exists, if it does then the lastScraped parameter is checked and compared with threshold so that Items which have just been scraped are not unnecessarily scraped again. Currently this threshold value is set to 24 hours.

The endpoints from this group depend heavily on Elasticsearch. There are services and wrappers to encapsulate operations such as searching, creating or adding of a new items. Thanks to the three endpoints, the scope of functionality in the Scraper and Web Services can be limited to only what is needed to fulfill their respective functions. Exposing these endpoints also paves the way for other internal or even external services to leverage the dataset of items collected by Banot in the future.

**Reports and Notifications**

Thanks to the database of Web Resources and Items, the system is now endowed with a growing dataset of Items from online bazaars and it allows the user to search through it. However, one of the main problems outlined in section 1.2 is that just having the data, while useful, is not good enough. Without the ability to be notified when a new item gets posted or updated, the users

\[^{7}\]More details on searching from the end-user’s perspective is provided in section 3.3
would need to manually run searches on regular basis to check whether their stolen item appeared in some online bazaar and was picked up by Banot. The endpoints in this group (see the list in figure 3.6) facilitate the functionality allowing users to manage their Reports and other parts of the system to execute periodical checks for new items which then trigger notifications going out to the users. This section describes the choice of a database to store the Reports in, the process of creating a new report, commenting on the different fields that need to be set up or are automatically generated. It goes into a bit more detail on how the name of each report is generated so that it is easy to remember. Finally it outlines how the Banot API sends the email notifications to end-clients.

Figure 3.6: Individual endpoints manipulating with Reports

GET v1/reports/[email]
GET v1/reports/[email]/[report-name]

POST /v1/reports
POST /v1/reports/[email]/[report-name]/pause

POST /v1/reports/notify
POST /v1/reports/[email]/[report-name]/notify

DELETE v1/reports/[email]/[report-name]

Similarly to Web Resources, Instructions and Root Pages, the Reports are also stored in the MySQL database and managed using NextrasORM. Since Reports do not have any relations to other entities, they could very well be stored in either of the two databases. The use of MySQL is more reasonable because there is no need to run full-text searches on Reports.

When creating a new Report the following properties must be specified:

- **Query String** - this Lucene query is ran against the items dataset to find out whether there are any new records which should trigger a notification.

- **Email** - is necessary for identification and for delivery of the notification.

- **Frequency** - defines how often the user want to be notified about new matches. The value represents number of hours between notifications.
Apart from these values a **name** is generated for every new entry. To allow one user, with one email, to created multiple reports, the email *together* with the name form a unique identifier of a report. This also makes the endpoints intuitive because GET v1/reports/[email] returns all reports for given user and v1/reports/[email]/[report-name] returns one specific record. Because the name will be something that the user needs to actively work with (it shows up in the notification and clients can also manage their report using its name) it is useful to use a human legible ID instead of an integer or UUID. The codebase thus includes a small utility which generates a random string in the following format:

```
[adjective]-[animal]-[integer between 0 and 99]
```

There are 131 adjectives, and 102 animal names, chosen so that they are short and easy to remember. In combination with the random integer there are 1,336,200 combinations in total. To enforce uniqueness there is also a compound unique key (email and report name) in the MySQL database. The utility was inspired by the JavaScript human-id package [25] and implemented in PHP.

To complete the list of Report’s properties there are three timestamps:

- **Created** - set automatically when new entry created.
- **LastNotified** - updated whenever there is an attempt to notify the user about new items. This value is used when searching for new records, only items added or updated after this date will be considered as new ones.
- **NextNotification** - calculated based on LastNotified and Frequency. It is used when searching for reports where notifications may be due.

Once a new report gets created a confirmation message is sent to the user. The email contains a summary of the report and the estimated time of the next notification. To check whether a notification should be sent, Banot relies on an external scheduler service to ping the notification endpoint (see 3.6). When this endpoint is called a query is executed to find all reports with the NextNotification date in the past, for each of these a query is sent to Elasticsearch to find any items which have been added or updated since the LastNotification date. Regardless or whether there are any new items both of these timestamps get updated. If there are new items a notification email is sent.

---

8This timestamp gets updated even when there are no new items and the email notification is not sent.
9For example the cron demon on Linux or the Cloud Scheduler in GCP
sent. The top ten items, sorted by relevance, get printed into the email with a button linking the user to the full report. The total number of new matches is added to the subject of the notification. The email also includes a link which pauses the notifications and a link to delete the report entirely. Deletion of the report has no impact on the items matched by the query from that report.

To deliver these notifications Banot uses the MailGun API. An SDK of this integrated as an Extension of the underlying Nette framework so that the Nette\Mailer service can be used consistently. The extension has three configuration parameters: domain, apiKey, and a boolean defining whether a EU or US instance of MailGun is being used. These parameters are then passed into the MailGun factory which creates an instance of a client capable of (among other things) sending emails. This new instance is then set to replace the default Nette\Mailer service. Nette Documentation provides more details on how emailing and extensions work within the framework.

Endpoints described in this section allow a client (authorized consumer of the API) to create, find, and delete a report. Reports are identified by an email and a human readable ID. Other endpoints enable the client to send notification or pause notifications on a specific report. A Nette Extension was built to make use of the MailGun service for email delivery.

---

Banot API is a pivotal component of the whole system. It is a RESTful service providing an interface through which the internal services communicate while also exposing its endpoints so that even external clients can tap into resources of this system. This service is built in PHP, the API is versioned using a section of the URL, it uses api-key for authentication and the OpenAPI standard is followed to generate documentation. The Web Resources, Instruction, Root Pages, Items, and Reports are all of the entities that are managed by this service with Items being stored in Elasticsearch and the rest in a MySQL database. Notifications are being sent out using the MailGun delivery service.

To simplify and standardize the implementation this API in the Web and Scraper services an SDK for PHP was built. The details of that library are being explored in the next section.

### 3.3 Banot SDK

SDK stands for Software Development Kit and it can take different shapes and forms in general terms it is a set of tools that allow creation of applications for

---

This is done by setting the NextNotification value of the report to null.
a certain software package. This section describes the Banot SDK which falls into this definition. It is a library which encapsulates and abstracts away the code needed to make a call to the Banot API. Instead of having to send a request to a URL and process the response the SDK handles these tasks and provides set of methods which can be called and the fact that an HTTP request is being sent in the background becomes an implementation detail.

The SDK is built in PHP and as such it can be used by other PHP applications. It is distributed as composer package and once published it can be required as a dependency by running:

```
composer require banot/banot-php
```

The fundamental design of this library is influenced by the mailgun/mailgun-php SDK, however the banot-php library provides two types of clients: A synchronous client and an asynchronous client. The user of this library can choose which one they want to use. The async client requests an instance of the LoopInterface from a PHP framework called ReactPHP and binds the execution of its operations into it. The sync client creates a new instance of an event loop, then it creates an instance of the async client, passing this new even loop as an argument. Methods of the sync client then simply "forward" the their arguments into their respective counterparts in the async client and block execution until the operation finishes, returning data (instead of a promise) back to the user.

Before the user can call a method to perform a specific task, for instance search for items, they need to first call a method which defines a “namespace”. Each namespace then contains a set of methods which can be called. Analogously to the client classes there are two types of these namespaces, one for the sync client and one for async. Methods of the sync methods return a value, typically a DTO which wraps the response, where as the async methods return an instance of a promise which resolves with the same value that the sync method would return. To ensure that the namespaces contain the same methods, both types of a given namespace implement the same interface. Figure shows how to crate an instance of the sync client and search for items.

This library requires PHP version of 7.4 and above. It also lists two libraries as its dependencies: clue/buzz-react is the chosen implementation of a HTTP client, and clue/block-react is need to block the execution in the sync client while waiting for a response from the API. Finally it relies on composer and the PSR-4 standard to autoload its classes.

---

11. The \Banot\SDK\Client class.
12. The \Banot\SDK\AsyncClient class.
3. Realization

Figure 3.7: Example use of the Banot SDK Client

```php
require 'vendor/autoload.php';
use Banot\SDK;

$client = new SDK\Client(BANOT_API_URL, BANOT_API_KEY);
$searchResult = $client->items()->search('bike');

echo "Found \${searchResult->getTotal()} items in total."

foreach ($searchResult->getItems() as $item) {
    echo $item->getTitle();
}
```

For testing the PHPUnit framework is used. The tests cover the hydration of the response DTOs, by creating a new one based on a JSON retrieved from the actual API and then testing the properties of the DTO. The shortcoming of this solution is that the API response is only copied instead of actually querying the API, so when the API structure changes, the copied response needs to be updated as well for the test to fail.

This SDK was developed to make it easier for the Web and Scraper services to interact with the Banot API. Nevertheless, it can very well be used by an external service as well. This library provides two clients: A synchronous one which only requires the URL and apiKey to be passed in as arguments and its methods return the final data. And an asynchronous client which requires an instance of an LoopInterface on top of the URL and apiKey and returns promises.

3.4 Scraper

The Scraper Service is the “workhorse” of the system. It is tasked with crawling and scraping of the online bazaar portals which are represented by the Web Resource entity. In the process of scraping it extracts information about items posted to these portals using the scraping Instructions which are defined for each Web Resource. The extracted data is then passed into the Banot API which handles the storage of this information. The unfortunate truth about web scraping is that most of the “processing” time is spent waiting for the content to be retrieved from the host. To get around this and make the process of scraping more efficient the Scraper service exploits asynchronous program-
3.4 Scraper

That way multiple requests can be processed (almost) at the same time. This section describes the process of web scraping using asynchronous PHP, it discusses the design patterns and techniques that come to play in this case. In the second part it is diving deeper into the codebase and it explains the build of this CLI application, the use of messaging queues, and the process of extracting the values from the raw HTML.

3.4.1 Process of scraping

The objective is to retrieve the URLs of the pages which present the detail of an item (the Detail Pages) and then extract information such as the item’s title, description, photo, price and more from that page. In order to get to these URLs, so called List Pages need to be processed first. In general a List Page is a paginated list of items where each item element contains a link to its Detail Page. Figure 3.8 shows an example of a List Page and a Detail Page with the elements of interest highlighted (these elements will come into play later in this section).

Even though both of these tasks, scraping of the List or the Detail pages, are very similar, it is easy to see that there will be significantly more Detail Pages than List Pages to process. This is because each List Page “produces” many Detail Pages. For this reason it is beneficial to be able to run the Scraper Service in two configurations. One is responsible for processing List Pages while the other is processing Detail Pages. To achieve this there are two commands that can be run when starting the service, each of them will initiate the service.

---

How many depends on the pagination settings.
3. Realization

in the desired configuration. The service, and this is true for all the other services in the system, is built as a docker image. In development environment the whole system runs using docker-compose, thanks to this arrangement it is possible to run multiple instances of the Scraper Service in each configuration. As it is described in section 4 there can be 10 instances of the Detail Page Scrapers for every one List Page Scraper, this however depends on many external factors. To process the pages in this way the scrapers rely on message queues to keep track of which pages need to be scraped. The RabbitMQ message broker is used to provide this functionality. As shown in figure 3.1 there are two distinct queues: a “LPQ” (List Page Queue) which holds the List Pages and a “DPQ” which holds the Detail Pages. A detailed description of how the scraper works when running in the two respective configurations is provided in sections 3.4.2 and 3.4.3. The rest of this section focuses on concepts that are universal for both of these use cases.

The Scraper service builds on top of the ReactPHP framework which

[..] is a low-level library for event-driven programming in PHP. At its core is an event loop, on top of which it provides low-level utilities, such as: Streams abstraction, async DNS resolver, network client/server, HTTP client/server and interaction with processes. Third-party libraries can use these components to create async network clients/servers and more.

Using this as its foundation the application is able to operate as a consumer (and a producer) of the messaging queues in an asynchronous fashion. This way the scraper can for example work on parsing one page while waiting for the content of another to be downloaded. The structure of the application and patterns used during implementation differ significantly when compared to a synchronous applications written in PHP. The use of Closures, Promises, and Event Loops is not common in these typical application application, however PHP has had a support for all of these techniques for quite a while as a proof of this the ReactPHP itself can be used with PHP 5.3 and above.

Reasonable people might disagree and argue that using PHP over, for example, Node.js for an application which is at its core asynchronous is not a good choice. According to benchmarks the general-purpose scripting language is performing comparably with Node.js when ran in the appropriate configuration. It is also important to take into account that PHP has came a long way from its origins with the 7th version introducing major performance im-

\[\text{Some of which are the network speed, hardware configuration, or the performance of the Web Resource.}\]

\[\text{PHP 5.3 was released on 30 June 2009.}\]
provements at the end of 2015. Furthermore, by using the same language as the Banot API and Web Services makes the use of a single SDK possible. Nonetheless, testing out how well can PHP perform at this task was one of the secondary motivations in pursuing this thesis.

Once the content is downloaded both of the scraper configurations create a new instance of the `Parser` class which requires the HTML of a page and a list of `Instructions` to extract the desired values. The instructions are retrieved from the Banot API Service using the asynchronous client of the SDK. Each instruction has the following properties:

- **Target** - determines whether this instruction applies for a List Page or Detail Page. Each configuration of scraper only works with the respective Instructions.
- **Name** - defines the naming of the extracted value. There is a set of predefined names that the scraper relies on.
- **Selector** - is used to locate an element or set of elements in the HTML DOM.
- **Type** - influences the way the values get extracted. The supported types enable the extraction of an attribute, link, or a text. For each of these there are plural variations (attributes, links, texts) which can be used to extract values from a set of elements.
- **Attribute** - is a property which can only be specified for instruction of the `attribute` and `attributes` types. It defines the name of the attribute whose value gets extracted.

There is also a plan to implement modifiers so that the extracted value can be formatted or processed in a different way before being passed over to the Banot API. Even though some components of the system, such as the ORM entities in the API service, already have the data structures in place, the implementation of the various modifiers within the Scraper Service is outside of the scope of this thesis.

Lastly, before moving onto the detailed description of how the List and Detail pages are processed, this is a list of few other important libraries which the scraper relies on:

- **symfony/console** - is used as a foundation for the build of the CLI.
- **symfony/dom-crawler** - is essential for the navigation of DOM and extraction of values.
• **bunny/bunny** - makes it easier to interact with RabbitMQ message broker.

• **clue/buzz-react** - is a simple, async PSR-7 HTTP client for concurrently processing any number of HTTP requests and as such it is used to send requests to the hosts and retrieve the content of the pages that need to be scraped.

### 3.4.2 Scraping List Pages

In relation to the message broker the List Page Scraper operates in multiple roles. It is a consumer of LPQ, processing one List Page after another. While also being a producer pushing succeeding List Pages into the queue. Lastly, it is also a producer to the DPQ since it is pushing the discovered Detail Pages into this queue.

When consuming from LPQ, each message contains one List Page which needs to be processed. The message is in plain text and follows this format:

```
[web_resource_name]|[list_page_url]
```

The *web_resource_name* is important so that the correct parsing instruction can be retrieved and the *list_page_url* is used to download the content of the page.

To process each List Page the Scraper needs to:

1. Request the parse Instructions from Banot API depending on the Web Resource to which the List Page belongs.
2. Fetch the HTML of the List Page from the host.
3. Extract the link to the next List Page by using the mandatory *nextUrl* instruction. This link is then pushed as a new message back into LPQ. When this link cannot be extracted the scraper concludes that it has reach the end of this collection and logs the information.
4. Extract links to the Detail Pages.
5. Query the Banot API, checking which of the extracted Detail Page URLs actually need to be scraped.
6. Push all the URLs that do need to be scraped into DPQ.
3.4.3 Scraping Detail Pages

In a similar fashion to 3.4.2 this section provides a breakdown of the steps needed to extract data from a Detail Page and pass them into the Banot API.

The Detail Page scraper acts only as consumer of DPQ, prefetching messages and sending back an acknowledgment of the message once the Detail Page is processed.

To process each Detail Page the Scraper needs to:

1. Request the parse Instructions from Banot API depending on the Web Resource to which the List Page belongs.
2. Fetch the HTML of the Detail Page from the host.
3. Extract the item details from the HTML.
4. Push the item into Banot API using the async client.
5. Send an acknowledgment to the message broker.

In step 3 when parsing there are two types of values that can be parsed. There are the “standard values” - title, photoUrl, description, price, published. While these values are not mandatory an attempt to extract them is always made and these values, with these specific names, are then used in throughout the system. On top of these, there are so called labels. These are any other values retrieved based on parsing instructions. Section 3.5 shows how are these values presented to the end-client withing the Web Service.

* * *

To summarize, the Scraper is the service responsible for the retrieval of the item data fro Web Resources. To work efficiently it is built using ReactPHP so that it can process pages asynchronously. The pages are divided into two groups, there are the List Pages which provide links to a set of Detail Pages which contain the desired information about items. These two groups of pages are processed separately and for that reason there are two distinct configurations in which the scraper can operate. To keep track of which pages need to be scraped two messages queues are used, one for the List Pages and the other for the Detail Pages. These queues are then operated by instances of the scraper service which functions both as a producer and a consumer of messages. Information extracted using the Instructions associated with a particular Web Resource is then passed into the Banot API which persists this data.
3. Realization

3.5 Web

The Web service is the least complex service in the system, its purpose is to provide a user friendly UI for the end-client to search through the dataset of items scraped by the Banot system. It also gives the user an option to turn on notifications by creating a report which monitors a given query and periodically notifies the user when new items matching this query are found. Last but not least, the user can view their created report and browse through the found items. This section describes the functionality from user’s perspective before providing an insight into the technologies and tools used for implementation.

3.5.1 Website screens

The landing page of the website explains basic information about Banot and steers the user towards the search screen or a form which lets the user find their existing report, figure 3.9 shows this form.

![Figure 3.9: Show Report Form](image)

The search screen is built to be minimalistic and simple to use, it consist of a search bar a button to execute search. When some items match the query the first ten most relevant results get listed other items can be retrieved by scrolling to the bottom of the list and pressing a button which loads ten more items. Each item is presented in a card components.

Figure 3.10 shows this component which provides information about the scraped item, when available it lists the title photo of the item, it prints the title and description of the item. On top of that there are two types of badges. The blue badges represent so called “standard parameters”. The Scraper Service either attempts to obtain these parameters for all items, that’s the case of the price and published parameters, or the parameter is generated by the scraper itself regardless of the values scraped from the Detail page, that’s the case with the last scraped badge. The gray badges on the other hand represent so called “labels”, as it is described in more detail in section 3.4, each Web Resource
can be associated with a set of instructions which are used to extract information from the Detail Pages, the instruction can extract information beyond the standard parameters, all such values are stored as a key-value pairs called labels. An example from figure 3.10 shows a label called “location” with the value of “Mladá Boleslav”\(^\text{16}\). The last scraped is the only badge which will be always present the other standard and and label badges will show up only when non-empty values were retrieved for the given item.

Finally, there is the “Open” button. When this button or the text is clicked the original Detail Page of the item hosted on the online bazaar platform opens up in a new tab of user’s browser.

To define a search the user can make use of the full Lucene syntax. The full description of this syntax can be found in \(^\text{24}\), however figure 3.11 shows examples of valid Lucene queries. The ability to search for terms, phrases, fields, ranges in combination with boolean operations, groping and relevance boosting gives an advanced user user a power tool to express exactly what they are looking for. At the same time the user does not even need to know about Lucene syntax to find useful results, just by typing in few words which get interpreted as terms the document containing these words will be matched and documents containing more of these terms will have higher relevance.

This advanced full-text search is possible thanks to Elasticsearch, the underlying search engine.\(^\text{17}\)

Once the user is satisfied with their search criteria they can click the “Turn on notifications” button and create a report. To create the report they need to fill in their email address and choose the frequency with which they want to be notified about new matches. The options are once a day (24 hours), once a week (168 hours), or ASAP which sets the frequency to 0 hours which means that the Report will be checked for new matches in every notifications

---
\(\text{16}\)A town in the Central Bohemian Region of the Czech Republic.
\(\text{17}\)See the \(\text{3.2.3}\) subsection for more details on how the Elasticsearch is used internally.
3. Realization

Figure 3.11: Examples of Lucene queries

# Search by terms
bike shimano

# Search by terms with boolean operators.
# The OR operator is the default one thus
# this query will return same results as the previous one
bike OR shimano

# Phrase search
"bike shimano"

# Search by field values
labels.location:"Mladá Boleslav"

# Boosting relevance of a term or phrase
# Documents containing "shimano" will be more relevant
bike shimano^4

# Range searches
# The following query find items with price 500 and above
price:[500 TO *]

# Gropping
# Find items which contain the terms "bike" and the
# phrase "shimano ultegra" in the description field
description:(+bike +"shimano ultegra")

cycle (see the Reports and Notifications subsection of 3.2.3 for more details
on how the reports get processed and how the notifications are sent).

The third and final use case of the Web service is when a user wants to revisit
a report they have already created. Whenever a new report is created a name
in the format of [adjective]-[animal]-[integer between 0 and 99] is generated
using their email and this name the users can access their reports. They can
see all the items matched by the query associated with this report and have
the option to pause the notifications or delete the report.
3.5. Technology & tools

Having described the functionality from user’s perspective, the next few paragraphs unpick the technical aspects of the website.

Combination of PHP and Nette framework was used to build this service. However, given that this service is the front-end of the system and it is doing very little apart from communicating with the Banot API the use of a front-end JavaScript framework such as React, Angular, or Vue would very likely be at least as good.

Nevertheless PHP 7.4 and Nette Framework 3.0 are used. The Banot SDK client is implemented into Nette as a DI extensions. This is a technique which lets a developer to add extra services to the Nette Dependency Injection Container from a configuration file. Through the extension it possible to configure the client: the api URL, api key, and the choice between synchronous and asynchronous client can be configured. In this case the synchronous client is used and the api URL and key are taken from respective environment variables.

Few elements of the website are broken down into components implemented using Nette components model. An example of this is the Item Card (presented in figure 3.10). Using components separates the code of this element into its own files, but most importantly it makes the element reusable. An example of this “reusability” is the ItemList component which is used on the search page and also on the page which shows an existing report to the user. Components in Nette can also contain other components which is something that the ItemList exploits as well by using the Item Card component.

Apart from DI extensions and components there are other ways in which the framework makes it easier to build and maintain the application, the last example provided here is the implementation of localization. Nette framework comes with a translator interface and a macro which makes it really easy to use translations in templates. However it does not provide the translator implementation out of the box, for this purpose the contributte/translation extension was used. Using the localization is beneficial for two reasons. Firstly it is the functionality it provides to users, the website can be viewed in multiple languages. And secondly it lets the developer to separate the texts of the UI from the code itself. The codebase contains keys which merely reference the strings. This application stores the strings in language specific neon files.

Nette and PHP are responsible for rendering the HTML which is then sent to the client. The Bootstrap 4 framework in combination with few JavaScript libraries and few lines of custom SCSS styles are used to format this HTML and style it into a responsive user interface. To make the interface more user
3. Realization

friendly and intuitive, vector illustrations provided by icons8.com as well as the Font Awesome icons are used. All of these assets are processed using webpack. More specifically a tool developed as part of the Symfony project called Webpack Encore is used. This tool is a simpler way to integrate Webpack into an application. It wraps Webpack, providing a clean & powerful API for bundling JavaScript modules, pre-processing CSS & JS and compiling and minifying assets \[35\]. Finally a tool called Yarn is used to manage the npm packages that the project depends on.

The Web service provides an interface for the users to interact with the dataset of items scraped by the Banot system. It allows the users to search through the items, set up notifications by creating a report and it lets the users view and manage these reports. It is built using PHP 7.4 and the Nette framework, relying on the Banot SDK to communicate with Banot API. Thanks to localization the users can view the website either in English or Czech. Bootstrap 4 is used to provide a clean and responsive look to the website.
Example use case & Testing

In this chapter everything from the Analysis and Realization chapters comes to gather and it is used in a real world scenario. The next few sections walk through the whole process starting with the creation of a new Web Resource with parsing Instructions and Root Pages. Then initiating the scraping via an API endpoint providing statistics and analysis of the scraping performance. After that the sections switch the perspective to the end-user and present how they can search through items, create and manage a report and receive periodic notifications when new items match the search criteria defined in the report.

The example used in this chapter presents a use case of scraping bike postings from two Czech-based online marketplaces (sbazar.cz and cyklobazar.cz). By the end of this chapter it should however be clear how this can be easily adapted to for example scrape motorbike postings from ebay.com.

4.1 Create a Web Resource

In order to start scraping the system needs to know what information it should extract and where is this information located. To provide the system with this information the entity of a Web Resource needs to be created. Attached to each Web Resources there are two types of entities - the Instructions and Root Pages. The Root Pages give the system a starting point from which can start crawling the online marketplace and eventually retrieve the HTML containing the information about the items posted there. The parameters of Instructions then give the system everything it needs to parse this HTML and extract the desired values.

A new resource can be created by sending a POST request to the /v1/resources endpoint. The body of this request is JSON containing the information about
4. Example use case & Testing

the Web Resource itself, as well as the definitions of the associated Instructions and Root Pages. Figure 4.1 shows an example of how the definition of the sbrazar.cz portal looks like. This endpoint is currently not implemented in the SDK, the main reason being that there is currently no service in the system which would provide a UI for Web Resource creation. See section 5.1 which proposes such service in the Ideas for further work chapter.

The resource can be deleted by sending a DELETE request to /v1/resources. To update the resource the original one needs to be deleted and subsequently a new one with the same name crated.

To follow along with the use case described in the introduction of this chapter another Web Resource called cyklobazar is created in the same fashion. Once these two entities are created, the instructions and root pages defined it is possible to initiate the scraping.

4.2 Run scraping

To initiate scraping, the POST /v1/resources/scrape request needs to be sent. The first one retrieves the Root Pages of all the Web Resources the latter does that only for a specific resource. Either way, these Root Pages are pushed into the List Page Queue which starts the process of scraping. There is a detailed description of this in sections 3.2.3 and 3.4.

In the use case which is being analyzed in this chapter both of the Web Resource had only one Root Page associated with them. The test was executed in an environment with following conditions:

- **Network** - 165 Mbps download, 178.3 Mbps upload, 20 ms latency
- **Hardware** - 16 GB RAM, Intel Core i7-8650U CPU @ 1.90GHz × 8
- **Number of List Page Scrapers** - 1
- **Number of Detail Page Scrapers** - started with 6, scaled up to 10 by the end of the experiment.

The scraping took 16 minutes and 10 seconds. In this period of time 788 List Pages were processed and 19,835 items indexed which gives the scraping speed close to 20.45 items per second. Including the List Pages the scraper was processing 21.26 pages per second.

---

18 The system runs in docker, the provided parameters are of the host machine.
During the scraping by far the highest memory consumption was registered by the Elasticsearch service and the highest CPU load by the Banot API service. In a production environment it would be reasonable to allow the api service to scale as well since there are multiple requests that this service needs to handle for each scraped item. Also in the current setup the Instructions are being requested for every item and List Page which is an unnecessary requests in most case and the number of these requests could be decreased significantly by caching the instruction in the Scraper service.

In a production with more computational resources and higher bandwidth the number items scraped per second could be increased to a point where the host might become overloaded or their throttling limits breached. Such situation might call for additional development.

With close to twenty thousand items in the dataset the next sections switch to the end-users perspective and describe the searching a notification aspects of the system.

### 4.3 Search through items

The details of how searching works and how it can be used by the end-user are thoroughly described in section 3.5. For the purposes of this section it is important to know that the Items are searched as text documents, the user can simply type in few words and all items that contain these terms in some of their fields will appear, the search engine determines the relevance of each item in the background and shows the most relevant searches first. Figure 3.11 shows examples of an advanced syntax which can be used to better define the search criteria, a complete overview of the syntax can be found in [24]. This section provides few examples of search queries which are applicable to use case of search for a bike which is being analyzed in this chapter.

Just for the sake of this example the user is looking for a Specialized bike which is fitted with the Ultegra groupset. The user could search for *specialized ultegra* which would however also return a lot non-bike items such as the ultegra groupset sold as a spare part because as long as at least on of these terms is present the item is deemed a match. To narrow down the search the user could use *+specialized +ultegra* which would only find items which contain both of these words. Or the user could include other words which are likely to occur in a classified add for a bike but not in one for a spare part.

Nevertheless this simple example shows, that while the full-text searching can be powerful there are some shortcomings and being able to at least categorize the items before running the text search would most likely lead to a decrease in the false positives. Section 5.2 describes a possible solution which could
leverage the photos of the items to help categorize them. This chapter is
only scratching the surface of the complexity involved in full-text searching
and searching in general. While there undoubtedly is a lot of room for im-
provement, the current solution provides a powerful tool to search through
the dataset of collected items.

4.4 Create a report and send notifications

There are two main problems when searching for a stolen item online, number
one there are potentially many different places where to look. Scraping and
indexing all of the items into one dataset solves that problem, now there is only
one place containing everything (in theory). That however does not address
the second problem which is that items are posted with high frequency which
makes it hard to keep up with what new has been added. This is why the
reports and notifications are important. A detailed description of how reports
work can be found in section 3.2.3, in this section an example report is created
examples of the confirmation and notification messages are shown.

To create a report the user defines a query, in this case the user searches for
any item that includes the word “specialized” and then clicks the “Turn on
notifications” button as shown in figure 4.2. After that they are taken to a
form where they fill in their email and chose how often they want to be notified
if something new is found. Once the report is created the user is presented
with a success page and a confirmation message is sent to the email. Figure
4.3 shows the confirmation message.

New matches are searched periodically and the notification is sent only when
at least on new item is found. Figure 4.4 shows an example of such notification.

This concludes the example use case, this section has gone through the whole
cycle from the creation of a Web Resource all the way to the notification
messages. This example worked with bikes and only two Czech-based portals,
however it is now hopefully easy to see that just by adding new Web Resources
with their specific instructions and Root Pages other online marketplaces and
different types of items could be scraped and search through as well.
4.4. Create a report and send notifications

Figure 4.1: Definition of the sbazar.cz as a Web Resource.

```json
{
    "url": "https://sbazar.cz/",
    "name": "sbazar",
    "instructions": [
        {
            "name": "nextUrl", "target": "list", "type": "link",
            "selector": ".c-prev-next
                .atm-button.c-prev-next__link[data-dot=next]"
        },
        {
            "name": "detailsUrls", "target": "list", "type": "links",
            "selector": "c-item-list .c-item__link"
        },
        {
            "name": "photoUrl", "target": "detail", "type": "attr",
            "selector": ".ob-c-carousel.ob-c-gallery__carousel
                img.ob-c-gallery__img:first-of-type",
            "attribute": "src"
        },
        {
            "name": "title", "target": "detail", "type": "text",
            "selector": "h1.p-uw-item__header"
        },
        {
            "name": "price", "target": "detail", "type": "text",
            "selector": "span.c-price b.c-price__price"
        },
        ...
    ],
    "rootPages": [
        {
            "url": "https://www.sbazar.cz/628-kola",
            "name": "Kola"
        }
    ]
}
```
4. Example use case & Testing

Figure 4.2: The search bar.

Figure 4.3: Confirmation message sent when report gets created.
4.4. Create a report and send notifications

Figure 4.4: Notification message sent when new matches are found.

51 New Items Found

Since the last notification (29. 07. 20) there are 51 new items which match the query **specialized**.

Top 5 most relevant matches

**Nabídka: Horské kolo Specialized**

Značka rámů: Specialized. 5001 a také Vittata: RockShox Rezon Silvers.zea 100x15 mm, vzhůru, lockout Zděné: 100 mm Železo: Shimano Deore Rapid-Fire, 2x10 Přesmykat: Shimano Deore Přeходováho; Shimano XT, 10 rychlostí

Rážeta: Shimano, 11-36...

View detail >

[... truncated]

View Complete Report

You can find your report using your email my@email.com and the name of the report Horsac octer-02.

https://banot.cz/report innv@email.com/horsac octer-02
Chapter 5

Ideas for further work

This section outlines areas which are outside the scope of this thesis but where it would be possible to make the current system easier to use or to add brand new functionality which would improve its capabilities of searching for stolen items online.

5.1 Graphical interface for adding new Web Resources

One shortcoming from the perspective of the end-user is that they have to work with what is given to them and have no ability to effect which Web Resources are scraped and what information is being collected about the items. Even though the system already exposes an API through which the Web Resources can be managed (see section 3.2.3), there is no user friendly interface which would allow a non-technical person to add a new Web Resource, list the Root Pages, and define the Instructions.

Defining the URL addresses of the Web Resource and the associated Root Pages is the easy part, the challenge of building such interface is the point where user needs to define the instructions which determine how is the item data extracted. For this to be easy to use the user should see a preview of a List Page and a Detail Page and define the instructions by clicking on the different elements which form the page. An inspiration could be drawn from an interface of a service called DataflowKit [36] which does exactly that. Figure 5.1 shows a screenshot of the DataflowKit interface.
5. Ideas for further work

Figure 5.1: DataflowKit “Build your Scraper” interface.

5.2 Google Cloud Vision

Currently the system focuses on leveraging only the text information. While it does process images which can help the user identify an item as they are browsing through the results of their search, it does not extract any information out of them. This is where a service such as Google Cloud Vision could prove very useful since it provides so called Vision API which takes an image as an input and can detect and extract information about entities in the image, this information is then returned as a response of this API. \[37]\]. Figure 5.2 shows an example of the labels returned when a photo of a bike gets passed in. While potentially extremely useful and easy to implement the cost of this service might be prohibitively high for a use at scale.

5.3 Production infrastructure

Within the scope of this thesis a fully functional system was developed, the different services of the system are built as docker images. To connect them together and allow scaling docker-compose is used. While this is an operational setup (see section 4) it does not allow for the system to be publicly available on the Internet. To achieve this a production infrastructure needs to be set up. Towards the end of implementation some limited experiments with Kubernetes and Google Cloud Platform were conducted, nevertheless this has not led to a successful production deployment. Apart from migrating the docker-compose configuration into Kubernetes configuration the production deployment might introduce some other challenges. With higher network bandwidth and more computational power available the scraper might be able to scrape at a speed which will cause it to be placed on a block list of the Web Resource, for that reason some throttling techniques might need to be put in place. Also
5.3. Production infrastructure

Figure 5.2: The Detect Labels functionality of Google Cloud Vision service.

balancing the load and setting up correct limits especially for the Banot API and the two scraper configurations might prove to be challenging as well.
Conclusion

The aim of this thesis was to address the problems associated with searching for stolen items online. In its theoretical part it describes the practice of selling stolen items online, so called E-Fencing, and by referencing other research it shows that this is problem of increasing severity while still being addressed only by a small number of publications. The thesis then analyzes existing solutions and tools that an average user has at their when searching for their stolen item on the web.

The inability to search for items from one place and the speed at which new items are being published to various online bazaars are established as the main problems. The technique of web scraping, full-text search, and giving the user an ability to set up periodic notifications is proposed as a solution that would improve the current state of things. Being able to expand the list of portals from which the items can be retrieved without the need to changed the source code is also stated as a requirement.

Based on the functional and non-functional requirements an architecture which consists of three main services was designed. A central REST API service was built to so that the other services can access the data they need. This central service also exposes the API for potential 3rd party services and it provides the means for sending of the notifications and initiation of scraping. To simplify integration of this API an SDK was developed as well. To crawl and process pages counting details about the items a Scraper service was built. To operate efficiently it is built to run in multiple instances which consume messaging queues and asynchronously process the pages that need to be scraped. Finally, to provide an interface for the end-user a Web service is built which allows the user to search through the dataset of items and set up notifications. The items dataset is stored in Elasticsearch which also serves as the search engine.
To test the performance of the system an experiment scraping two online bazaars was performed. During the 16 minutes and 10 seconds of the experiment over twenty thousand pages were visited and the total of 19,835 items indexed into the dataset resulting in the scraping speed of 20.45 items per second. These results are dependent on the environment in which the scraper runs and it is assumed that a production environment with higher bandwidth and more computational resources could reach even better numbers.
Acronyms

API  Application Programmable Interface
ASAP  As Soon As Possible
BANOT  Bazzar Notifications
CLI  Command Line Interface
DI  Dependency Injection
DNS  Domain Name Server
DOM  Domain Object Model
DPQ  Detail Page Queue
DTO  Data Transfer Object
ERD  Entity Relationship Diagram
FR  Functional Requirement
GCP  Google Cloud Platform
HTML  Hypertext Markup Language
HTTP  Hypertext Transfer Protocol
JSON  JavaScript Object Notation
LPQ  List Page Queue
NFR  Non-functional Requirement
A. ACRONYMS

**PHP**  PHP Hypertext Processor

**PSR-4**  4th PHP Standard Recommendation

**PSR-7**  7th PHP Standard Recommendation

**REST API**  Representational State Transfer API

**SDK**  Software Development Kit

**UI**  User Interface

**UUID**  Universally Unique Identifier
Bibliography


Appendix B

Contents of enclosed CD

The thesis is also available from

- https://github.com/TonyVlcek/banot
- http://banot.cz

api..............the directory with source code of the Banot API service  
banot-php.......the directory with source code of the Banot SDK library  
compose ...........the directory with the docker-compose configuration  
scraper ..........the directory with source code of the Scraper service  
web..................the directory with source code of the Web service  
thesis.pdf.................the thesis text in PDF format