



Assignment of master's thesis

Title:	Discord bot for Czech real estate market monitoring
Student:	Bc. Vojtěch Drška
Supervisor:	Ing. Ondřej Guth, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

Design and implement software for periodically scraping real estate related data in the Czech Republic from at least three web portals where one does not provide an API that returns directly the data of interest. Design and implement a Discord Bot which will provide its users with fresh data based on their interests and preferences. Deploy the whole application into a cloud environment in a scalable way.



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Master's thesis

Discord bot for Czech real estate market monitoring

Bc. Vojtěch Drška

Department of Software Engineering
Supervisor: Ing. Ondřej Guth, Ph.D.

June 29, 2023

Acknowledgements

I would like to thank Ing. Ondřej Guth, Ph.D. for supervising this thesis, his time, his effort, his suggestions and overall for an amazing and pleasant cooperation. I would also like to thank my parents for their endless support throughout my studies for which I am eternally grateful.

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 June 29, 2023

.....

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Vojtěch Drška. 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

Drška, Vojtěch. *Discord bot for Czech real estate market monitoring*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Tato diplomová práce se věnuje návrhu a implementaci softwaru sloužícího k získávání dat týkajících se nabídek na českém realitním trhu. Dále se tato práce zabývá návrhem a implementací Discordového bota, sloužícímu k notifikování uživatelů o nabídkách, které je zajímaví.

Klíčová slova Discord, bot, REST, API, Java, Python, nemovitosti, monitorování

Abstract

The subject of this master's thesis is to design and implement software used for extracting and parsing data related to Czech real estate market. Another objective of this thesis is to design and implement a Discord bot, capable of sending notifications to its users, containing information about real estate listings of their interest.

Keywords Discord, bot, REST, API, Java, Python, real estate, monitoring

Contents

Acknowledgments	1
1 Analysis	3
1.1 Analysis of existing solutions	3
1.1.1 Realitní hlídač pes	3
1.1.2 realitymon	3
1.1.3 inforeality	4
1.1.4 Conclusion	5
1.2 Requirements engineering	5
1.2.1 Functional requirements	6
1.2.1.1 F1: Subscribe to a location	6
1.2.1.2 F2: Subscribe to a location with filters	6
1.2.1.3 F3: Edit existing location subscription	7
1.2.1.4 F4: Cancel subscription	7
1.2.1.5 F5: List all user's subscriptions	7
1.2.1.6 F6: Send notification about new listing to sub- scribed users	7
1.2.1.7 F7: Send notification about price change to subscribed users	7
1.2.1.8 F8: Send notification about listing removal to subscribed users	7
1.2.1.9 F9: Manual scraper start	8
1.2.1.10 F10: Application health status	8
1.2.1.11 F11: Data updates	8
1.2.1.12 F12: Date & time of last update	8
1.2.2 Non-functional requirements	8
1.2.2.1 NF1: Responsiveness	9
1.2.2.2 NF2: User friendliness	9
1.2.2.3 NF3: Frequency of data scraping	9

1.2.2.4	NF4: Ethical load on scraped servers	9
1.2.2.5	NF5: Secured access to scrapers	9
1.2.2.6	NF6: Availability	9
1.2.2.7	NF7: Cost efficiency	10
1.2.2.8	NF8: Deployment automation	10
1.2.3	Use cases	10
1.2.3.1	UC1: Subscribe to location	11
1.2.3.2	UC2: Adding new subscription with filters	12
1.2.3.3	UC3: Updating an existing subscription	12
1.2.3.4	UC4: Removing existing subscription	13
1.2.3.5	UC5: Scraping new listings	13
1.2.3.6	UC6: Scraping price changes	14
1.2.3.7	UC7: Identifying removed listings	14
1.2.3.8	UC8: Manually triggering scraping of new data	15
1.2.3.9	UC9: Health status check	16
1.2.3.10	UC10: List active subscriptions	16
1.3	Domain model	16
1.3.1	Subscription	17
1.3.2	RealEstate	17
1.3.3	Address	18
1.3.3.1	Hidden location	18
1.3.3.2	Old listings	18
1.3.3.3	Same real estate, different portals	19
1.3.4	Channel	19
2	Design	21
2.1	Database	21
2.1.1	Data model	21
2.1.2	Real estate listings	21
2.1.2.1	Address	23
2.1.2.2	Suburb Prague	23
2.1.3	Database type	23
2.1.4	RDBMS	25
2.2	Scrapers	25
2.2.1	Programming language	25
2.2.1.1	Python	25
2.2.1.2	JavaScript (Node.js)	26
2.2.1.3	Ruby	27
2.2.1.4	C/C++	27
2.2.1.5	Conclusion	28
2.2.2	Architecture	28
2.2.2.1	Server-less/Lambda functions	28
2.2.2.2	Cron jobs	31
2.2.2.3	Discord bot	32

2.2.2.4	Conclusion	33
2.2.3	Commands	33
2.3	Discord bot - client side	34
2.3.1	Commands	34
2.3.1.1	Creating a subscription	35
2.3.1.2	Updating a subscription	35
2.3.1.3	Deleting a subscription	36
2.3.1.4	Viewing subscription details	37
2.3.1.5	Health-check	37
2.3.2	Programming language	37
2.3.2.1	C/C++	37
2.3.2.2	Java	37
2.3.2.3	Conclusion	38
2.3.3	RESTful API	38
2.3.3.1	Programming language & framework	38
2.3.3.2	Endpoints	38
2.4	Cloud provider	39
2.5	High-level architecture	40
2.5.1	Deployment	40
2.5.2	Managing listings - Scrappers and REST API	41
2.5.3	Managing subscriptions - Discord bot and REST API	41
2.5.4	Obtaining user info - Discord bot and REST API	41
2.5.5	Reverse geocoding - Nominatim and Scrappers	41
3	Implementation	47
3.1	Scraping Bot	47
3.1.1	Sreality scrapping	47
3.1.2	Scraping new listings	47
3.1.2.1	Scraping data from search results	47
3.1.2.2	Scraping data from listing details	48
3.1.2.3	Conclusion	49
3.1.3	Address scrapping	49
3.1.3.1	httpx vs requests	49
3.1.3.2	Reverse engineering of the API	49
3.1.4	Reality Idnes scrapping	50
3.1.4.1	Addresses	50
3.1.5	Bezrealitky scrapping	51
3.1.5.1	Scraping listing's details	51
3.1.6	Dependency Injection	51
3.1.7	SQLite as cache	52
3.2	Client bot	52
3.2.1	Sending images	52
3.2.2	Modals vs Views	53
3.2.3	Regions and municipalities	54

3.3	REST API	55
3.3.1	Architecture	55
3.3.2	Model conversion placement	56
3.3.3	ModelMapper	57
3.3.4	Hibernate Validator	57
3.3.5	Access control	57
3.3.6	Price change	58
3.3.7	Deriving missing parts of addresses from existing data	58
3.4	AWS	59
3.4.1	Multiple SQS queues VS. Message Groups	59
3.4.2	ECS Task definition	59
3.4.3	ECS Service	60
3.4.4	Deployment	60
4	Testing	63
4.1	Types of used tests	63
4.1.1	Unit tests	63
4.1.2	User tests	63
4.2	Client bot	63
4.2.1	User testing	63
4.2.1.1	Create subscription	64
4.2.2	Update subscription	65
4.2.3	View subscription details	65
4.2.4	Remove subscription	65
4.2.5	Verify the bot is working	66
4.3	REST API	66
5	Conclusion	67
5.1	Visions for the future	68
	Bibliography	69
A	Acronyms	75

List of Figures

1.1	realitymon - No results	4
1.2	Domain model	20
2.1	Data model	24
2.2	Client bot - create form - basic	35
2.3	Client bot - create form - filters	36
2.4	Client bot - subscription embed	42
2.5	Client bot - filled out forms	43
2.6	Client bot - subscription selection	44
2.7	Client bot - command overview	44
2.8	High-level architecture	45
2.9	High-level architecture with S3 bucket	46
3.1	Sreality house search results	50

List of Tables

1.1	Application features	6
1.2	Relations between use cases and functional requirements	17

Introduction

Nowadays, it is hard to buy a real estate at a reasonable price in the Czech Republic [1]. Therefore, once a good offer appears, it is crucial to respond fast as the demand for such offers is large and usually on the first come, first served basis. Because of that, frequently monitoring real estate portals for new offers is necessary, but at the same time very expensive time wise. That is exactly where our application is supposed to step in.

The goal of this thesis is to design and implement an application that will allow its users to efficiently monitor the Czech real estate market for their desired properties. The application will be composed of multiple parts. These parts are namely client bot, REST API and scrapers. The scrapers will be responsible for fetching and parsing data from multiple real estate portals. This data will then be sent to the REST API, which will save them to the database. It will also send details of notification details to a queue. From this queue, the notification details will be picked up by the client bot and transformed to notifications in the form of Discord messages. These notifications will then be sent to the interested users.

The thesis is split into five chapters. The first chapter 1 discusses the existing solutions. It also defines the functional and non-functional requirements as well as the use cases. The second chapter 2 contains discussions about selected technologies, designs of UI and the high-level architecture and the use of cloud services. Chapter number three 3 is devoted to implementation and discusses the problems that we came across. It also describes some of the solutions in a greater detail. The fourth chapter discusses the tests used in different parts of the application and types of used tests in general. Finally, the fifth chapter 5 concludes and discusses how we were able to achieve the goals of this thesis and accomplish the requirements from chapter one 1. It also describes some of the possibilities and directions in which the development of the application might go in the future.

Analysis

1.1 Analysis of existing solutions

1.1.1 Realitní hlídač pes

Realitní hlídač pes is an application that monitors real estate listings from multiple different web portals. It allows the user to specify the town/city of interest. The street may also be specified, but only for Prague and Brno. The user may also specify the radius around the chosen locality. If new listing appears the user is notified via an email.

The software notifies the users only about new listings, however it does not notify them about events such as price changes or listings getting taken down / properties being sold.

Looking at the pricing and the terms of different subscriptions the applications seems to be targeting mainly people and not real estate agencies. The terms of the subscriptions are also rather strict. The cheapest subscription costs 56,-Kč/month per monitor however to get this price you need to run at least three monitors and run them all for a year. The most expensive option costs 99,-Kč/month per monitor and while it does not require you to run multiple monitors it does require you to run the monitor for at least a three-month period. Although the cost of the subscriptions is quite reasonable the conditions and mainly the fixed time period is not very flexible and might deter some users from using the application [2].

1.1.2 realitymon

realitymon [3] is a web application providing an overview of properties for sale on different web portals all over the Czech Republic. The application states that it supports the following web portals:

- Sreality
- Bezrealitky

1. ANALYSIS

- Idnes
- Reality.cz
- viaReality

However, after examining and trying out the application it seems like only data from Sreality and Reality.cz are available. If the user chooses any other of the portals, the map displaying the listed properties is empty as we can see in the figure 1.1. It is possible that the application will get fixed or will start supporting the other portals in the future as the web application states that it is still in the development, but at this point in time only the two mentioned web portals seem to be working.

The application also does not enable the users to obtain notifications of any kind so it only works as a sort of an aggregator of different web portals. Therefore, it only eliminates the need for checking multiple different web portals while the need for periodic manual check remains.

As for the frequency of updates, the data displayed by the application are refreshed once a day. This could be an issue for some of the potential users because, in the worst case, they could be informed about a newly listed property with a 24 hour delay. This might allow someone else to show their interest in the property sooner than them.

The use of the application is free and there are no paid options available [3].

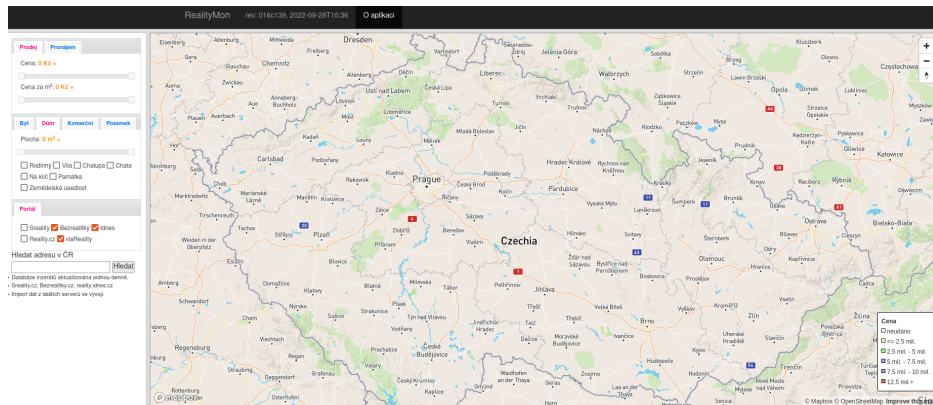


Figure 1.1: realtymon - No results

1.1.3 inforeality

inforeality is a real estate listing monitoring software. It monitors tens of web portals and provides fresh data every 1-3 minutes. The application allows you to set filters based on age of listings, type of real estate, location and a few more. The website which describes the software also states that the

information are available either through email or through an application. It unfortunately does not state whether the application is in a form of web or desktop application.

It was attempted to research this software further as the website that represents it seems rather dated and so it is not sure whether the software is actually still working, but to no avail. Trying the application is also not an option as the only way of obtaining the free trial is requesting it through a form which requires more personal information than we are comfortable giving to the company.

Regarding the pricing, there are multiple different subscriptions which range from roughly 1200,- Kč/month to 8500,-Kč/year.

It is also worth noting that the target audience of the software are mainly real estate agencies and therefore it might not be suitable for individuals [4].

1.1.4 Conclusion

From the three described applications, the inforeality [4] seems to be the closest one to the application that is going to be developed in this masters thesis. The key differences are that inforeality focuses mainly on the real estate agencies, is quite expensive for a "casual user" and does not provide discord support.

Realitní hlídač [2] is similar to our application in terms that it is affordable and targets casual users instead of real estate agencies. Where it differs is the lack of the following features:

- flexibility regarding the already set up monitors
- notifications based on change of price or a listing being removed
- discord support

realitymon [3] seems to be only a listing aggregator which currently supports only two web portals. Apart from working with other web portals it does not have much of a overlap with our application in terms of functionalities and features.

The overview of supported features by different applications can be seen in the table 1.1. There is one value regarding our application that should be shortly addressed. The exact refresh rate of our application will be determined later in the implementation chapter 3 as it will be affected by the behavior of web portals when scraping as well as the method of scraping that will be chosen.

1.2 Requirements engineering

This section will discuss the functional and non-functional requirements that the application must satisfy. The requirements will describe all important

Table 1.1: Application features

	Realitní hlídač pes	inforeality	realitymon	Our Application
Free version	Yes	No	Yes	Yes
Refresh rate	n/a	1-3 min.	24h	5-15min.
Number of portals	n/a	10+	2	3
New listing notif.	Yes	Yes	No	Yes
Price change notif.	No	Yes	No	Yes
Removed listing notif.	No	Yes	No	Yes
Email notif.	Yes	Yes	No	No
Discord notif.	No	No	No	Yes

functionalities and features that must be available to the end users. Different use cases and mapping of functional requirements to them will also be described in this section [5][6].

1.2.1 Functional requirements

Functional requirements define what features have to be implemented in order to allow the users achieve their goals. They typically describe the behavior of an application. Stating functional requirements and keeping them in mind during development also helps us make sure that the system is doing what it is supposed to do and therefore brings the expected value to its users.

1.2.1.1 F1: Subscribe to a location

The application must allow the user to subscribe to a location using a Discord command.

1.2.1.2 F2: Subscribe to a location with filters

The application must allow the user to subscribe to a location and add filters for the following attributes:

- real estate type
- number of rooms
- land dimensions
- real estate dimensions
- energetic efficiency
- state of the real estate
- min. price

- max. price

using a Discord command.

1.2.1.3 F3: Edit existing location subscription

The application must provide user with the option of editing an existing subscription to a location. This will allow the user to edit filters on all the attributes specified in requirement F2 1.2.1.2.

1.2.1.4 F4: Cancel subscription

The application must allow the user to cancel an existing subscription.

1.2.1.5 F5: List all user's subscriptions

The application must allow the users to list all the locations that they are subscribed to. This listing must also include all filters described in the requirement F2 1.2.1.2 that are active. As one of the reasons for listing the active subscriptions is to perform some action on them, all the subscriptions should include some form of ID, so they can be later referred to in Discord bot commands.

1.2.1.6 F6: Send notification about new listing to subscribed users

The application must send a notification that new listing appeared to all users that have a subscription matching the newly found listing. In addition to that, each separate notification must take into count the specific filters setup by each user and adhere to them.

1.2.1.7 F7: Send notification about price change to subscribed users

The application must send a notification when price changes for an existing listing to all its subscribers. In addition to that, each separate notification must take into count the specific filters setup by each user and adhere to them.

1.2.1.8 F8: Send notification about listing removal to subscribed users

The application must send a notification that listing was removed to all its subscribers. In addition to that, each separate notification must take into count the specific filters setup by each user and adhere to them.

1.2.1.9 F9: Manual scraper start

The application must provide an option of manually starting data scraping. This feature must be available exclusively to the administrator of the application as its unsupervised and parallel usage could lead to issues as described in the requirement NF5 1.2.2.5.

1.2.1.10 F10: Application health status

Any user must be able to check the health of the application at any point in time. This may be done simply as a *ping* discord command.

1.2.1.11 F11: Data updates

The application must periodically refresh its data in order provide the users with relevant information.

1.2.1.12 F12: Date & time of last update

The application should, on request, provide the users with last date and time when the data was refreshed/updated.

1.2.2 Non-functional requirements

In contrast to functional requirements, non-functional requirements do not describe the features or behavior of a system, but rather how these features should be performed. Non-functional requirements are typically focused around the following areas:

- Performance — speed, scalability, responsiveness
- Security — authorization, protection of sensitive data, detection of security breaches
- Usability — ease of accomplishing users goals, UI user friendliness
- Reliability — behavior is as expected, ability to function consistently and reliably
- Maintainability — proper testing, code quality, easy to maintain
- Availability — percentage of successful requests, up-time
- Portability — compatibility with different environments, ease of deployment to new environment

Even though non-functional requirements do not define the behavior of the system, they play a huge role in its success. This is mainly due to the fact that the areas described above are all of great importance and they determine the overall quality of the system [7][8].

1.2.2.1 NF1: Responsiveness

Even though Discord bot does not feature any GUI, the bot should still "feel responsive" by providing the users with clear and deterministic responses indicating the status/success of their command.

1.2.2.2 NF2: User friendliness

The commands supported by the bot should be easy to use and intuitive. In case any form needs to be filled out by the user, the fields and its contents should be clearly explained to the user.

1.2.2.3 NF3: Frequency of data scraping

The data should be scraped as often as possible. This is important mainly for the monitoring of newly added listings as infrequent updates could cause the users to miss out on a deal. This requirement should, however, be in compliance with requirement NF4 1.2.2.4.

1.2.2.4 NF4: Ethical load on scraped servers

The frequency of scraping data must be reasonable and not cause an excessive load on the scraped servers. Even though we want to keep the users updated with fresh data as frequently as possible as described in the requirement NF3 1.2.2.3, behaving ethically correct and not causing any harm to the scraped servers is of priority.

1.2.2.5 NF5: Secured access to scrapers

Manual access to the scrapers and their triggering should be an action reserved only to the administrator of the application. Not restricting the access could lead to issues such as:

- crashing the application when many users trigger the scrapers in parallel
- creating heavy load on the servers being scraped which would be against the requirement NF4 1.2.2.4

1.2.2.6 NF6: Availability

The availability of the application must be ensured by using a reliable production environment with stable internet connection. This is why the application should be deployed to cloud as majority of cloud providers provide SLAs guaranteeing up-time north of 99% [9].

1.2.2.7 NF7: Cost efficiency

The whole application should be developed with cost efficiency in mind. This applies not only to the architecture/design of the application, but also, and mainly, to the choice of cloud provider and used cloud services. With that said, the choices made in order to save resources should in no way hinder the application's safety or usability.

1.2.2.8 NF8: Deployment automation

In order to make the process of deploying the application fast and reliable, CI/CD pipelines should be created in order to automate the deployment. The pipelines should be able to both deploy new versions of application to existing servers as well as deploy the application to a new environment.

1.2.3 Use cases

Use cases are used to define goals that need to be accomplished in the system. The main parts of each use case are typically the following:

- Actor — the entity that is going to be interacting with the system. This may be a user as well as administrator or, in some cases, time
- Goal — what is supposed to be achieved by completing the defined series of operations described in the scenario
- Scenario — describes the sequence of operations and interactions that need to be done in order to reach the goal of the use case

Each use case should also have a unique number/id and a unique name so it can be easily referred to. There are also multiple optional components that the use case may contain.

- Pre-conditions — Pre-conditions specify in which state the system has to be (what needs to happen) before the scenario specified in the use case can be executed
- Alternative path — The scenario included in the use case is usually considered to be the main success scenario. In some use cases, however, there might be an alternative sequence of interactions with the system which still leads to the end goal. This sequence is then described as an alternative path
- Exception — During the execution of a scenario, exceptions might happen. These can be in the form of user canceling a form or passing in an incorrect input

- Post-conditions — As there may be multiple ways in which the use case might end, there also can be multiple different states in which the system may end up in, depending on how the execution of the use case ended

In addition to the previously discussed attributes, our use cases will also contain the list of functional requirements, that are used in the scenario of the respective use case.

An overview of all the relationships between the individual use cases and functional requirements can be viewed in the following table 1.2.

The point of specifying different use cases is, that it helps us outline the ways that the users will be interacting with the systems. It will also help us understand and document the different dependencies between requirements and the different states that the system might get into while being interacted with [10][11][12].

1.2.3.1 UC1: Subscribe to location

Used requirements:

F1: Subscribe to a location 1.2.1.1

F2: Subscribe to a location with filters 1.2.1.2

F5: List all user's subscriptions 1.2.1.5

Actor: Any user

Goal: Subscribing to location

Pre-conditions: The user is either in chat with the Discord bot, or is in a channel that the bot is a member of.

Scenario:

The user enters a command, indicating that they want to create a new subscription. The bot receives the command and responds with a message or series of messages allowing the user to specify all the attributes upon which the listings can be filtered. After receiving the response(s), the user specifies only the location and sends it to the bot. The bot processes the response and subscribes the user to the selected location. After that the bot sends a response to the channel, from which it was contacted by the user, indicating that the action was successful and listing all currently active subscriptions including filters if applicable.

Exceptions:

The location specified by the user is invalid. The bot therefore sends a response to the channel, from which it was contacted by the user, indicating that the action was unsuccessful and listing all available locations.

Post-conditions:

In case the scenario took place and the action was completed successfully, the user is now subscribed to the requested location.

In case of the alternative path, the state of the system remains the same as the passed in location was invalid and therefore no new subscription was created.

1.2.3.2 UC2: Adding new subscription with filters

Used requirements:

F2: Subscribe to a location with filters 1.2.1.2

F5: List all user's subscriptions 1.2.1.5

Actor: Any user

Goal: Creating a subscription with multiple options/filters

Pre-conditions: The user is either in chat with the Discord bot, or is in a channel that the bot is a member of.

Scenario:

The user enters a command, indicating that they want to create a new subscription. The bot receives the command and responds with a message or series of messages allowing the user to specify all the attributes upon which the listings can be filtered. After receiving the response(s), the user specifies the filters that they are interested in and submits them. The bot processes the messages and subscribes the user to the selected location with all the specified filters. After that the bot sends a response to the channel, from which it was contacted by the user, indicating that the action was successful and listing all currently active subscriptions including filters if applicable.

Post-conditions:

In case the scenario took place and the action was completed successfully, the user is now subscribed to the requested location.

1.2.3.3 UC3: Updating an existing subscription

Used requirements:

F3: Edit existing location subscription 1.2.1.3

F5: List all user's subscriptions 1.2.1.5

Actor: Any user

Goal: Updating an existing subscription

Pre-conditions:

The user has an existing subscription

The user is either in chat with the Discord bot, or is in a channel that the bot is a member of.

Scenario:

The user enters a command, indicating that they want to update an existing subscription. The bot receives the command and responds with a drop-down

menu containing all the currently active subscriptions. After receiving the drop-down menu, the user selects the subscription that they want to update. The bot processes the response and sends the user a message or series of messages similar to the one in UC2 1.2.3.2. Compared to the responses sent in UC2 1.2.3.2 there is one small difference. That is, all the messages (modals, drop-downs, forms) that are sent to the user are already pre-filled with the values of the already existing subscription. After the user sends the filled out messages back to the bot, they are processed and the existing subscription is updated.

Post-conditions:

In case the scenario took place and the action was completed successfully, the user is now subscribed to the requested location.

1.2.3.4 UC4: Removing existing subscription

Used requirements:

F4: Unsubscribe from a location 1.2.1.4

F5: List all user's subscriptions 1.2.1.5

Actor: Any user

Goal: Removing an existing subscription

Pre-conditions: The user must have an existing subscription

Scenario:

The user enters a command, indicating that they want to remove/cancel an existing subscription. The bot receives the command and responds with a drop-down menu, containing all the active subscriptions of the respective user. The user picks the subscription that they want to cancel from the drop-down menu, which passes it back to the discord bot. Once the bot receives the subscription, it removes it from the system. After doing so, it sends a response to the channel of origin, indicating the successful cancellation of given subscription and listing all the remaining subscriptions.

Post-conditions:

In case the scenario was completed successfully, the subscription selected by the user is now removed from the system.

1.2.3.5 UC5: Scraping new listings

Used requirements:

F7: Send notification about new listing to subscribed users 1.2.1.6

F12: Data updates 1.2.1.11

Actor: Time

Goal: Finding newly added listings to portals

Scenario:

Once the scrapers are triggered after a specified duration, they scrape all the supported portals for newly added listings. In case new listing was found, its details are scraped and saved to the database. At the same time, the information that new listing was found is passed to the bot alongside with its details. The bot then checks which users are interested in this listing, based on the existing subscriptions. After determining which users are interested, the bot sends a message to each one of them, stating that a new listing was found as well as details of the listing.

Post-conditions:

In case the scenario was completed successfully, the system database now contains the most recently added listings to the supported portals. Also, a message to the subscribed users was sent.

1.2.3.6 UC6: Scraping price changes

Used requirements:

F8: Send notification about price change to subscribed users 1.2.1.7

F12: Data updates 1.2.1.11

Actor: Time

Goal: Finding for which listings has the price changed

Scenario:

Once the scrapers are triggered after a specified duration, they scrape all the supported portals in order to determine for which listings the price changed. In case a price change is found, the scrapers save it to the database. At the same time, the information that the price changed for some listing is passed to the bot alongside with the updated listing details. The bot then checks which users are interested in this listing, based on the existing subscriptions. After determining which users are interested, the bot sends a message to each one of them, stating that price was changed as well as updated details of the listing.

Post-conditions:

In case the scenario was completed successfully, the system database now contains up to date information regarding the price changes. Also a message describing the price change was sent to subscribed users

1.2.3.7 UC7: Identifying removed listings

Used requirements:

F9: Send notification about listing removal to subscribed users 1.2.1.8

F12: Data updates 1.2.1.11

Actor: Time

Goal: Identifying which listings were removed

Scenario:

Once the scrapers are triggered after a specified duration, they scrape all the supported portals in order to determine which listings were removed. In case some listing is missing, compared to the previous scraping run, the scrapers save that information to the database. At the same time, the information that some listing was removed is passed to the bot alongside with its details. The bot then checks which users are interested in this listing, based on the existing subscriptions. After determining which users are interested, the bot sends a message to each one of them, stating that an existing listing was removed, alongside with its details.

Post-conditions:

In case the scenario was completed successfully, the system database now contains up to date information regarding which listings were removed. Also a message notifying that a listing was removed is sent to all interested users.

1.2.3.8 UC8: Manually triggering scraping of new data

Used requirements:

F10: Manual scraper start 1.2.1.9

Actor: System administrator

Pre-conditions: The system administrator is either in chat with the Discord bot, or is in a channel that the bot is a member of.

Goal: Manually starting the scraping process

Scenario:

The system administrator sends a command, indicating that they want to scrap the real estate portals to the scrapper bot. Once the bot receives the command it checks that the user sending the command is actually the system administrator a therefore is authorized to perform the command. Once that it is verified, the bot responds with a modal that allows the administrator to pick which portals and what data (new listings, changed prices, removed listings) do they want to scrap. Once the administrator submits the modal, the bot starts scraping the requested portals for specified data. The bot will also inform the administrator about the starts and ends of the requested scraping jobs via a series of messages.

Post-conditions:

In case the scenario was completed successfully, the system database now contains up to date information regarding the data specified by administrator. Also a message notifying that a change in data was found is sent to all inter-

ested users.

1.2.3.9 UC9: Health status check

Used requirements:

F11: Application health status 1.2.1.10

F13: Date & time of last update 1.2.1.12

Actor: Any user

Pre-conditions: The user is either in chat with the Discord bot, or is in a channel that the bot is a member of.

Goal: Checking whether the bot is running

Scenario:

The user send a ping to the bot. After the bot receives the command, it responds with a message, indicating that it is alive and ready to take any commands, and stating the last time when the data was refreshed.

1.2.3.10 UC10: List active subscriptions

Used requirements:

F5: List all user's subscriptions 1.2.1.5

Actor: Any user

Pre-conditions: The user is either in chat with the Discord bot, or is in a channel that the bot is a member of.

Goal: Retrieving list of all active subscriptions

Scenario:

The user sends a command, indicating that they want a list of existing subscriptions as the response, to the bot. After the bot receives the command, it responds with a message containing the list of all the user's active subscriptions alongside with their filters if applicable.

1.3 Domain model

Domain model is a type of class diagram. Its purpose is to represent different entities present in the concerned domain and the relationships between them. Domain model should also be platform independent. Entities in the

Table 1.2: Relations between use cases and functional requirements

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
UC1	X	X			X							
UC2		X			X							
UC3			X		X							
UC4				X	X							
UC5						X					X	
UC6							X				X	
UC7								X			X	
UC8									X			
UC9										X		X
UC10					X							

domain model are represented as classes which are usually composed of name, attributes and methods.

In case of our domain model, the attributes and methods are missing. The reason for that is, that while we now with a great level of certainty what the entities in our domain are, their attributes are mostly dependent on the data sources that the data are scraped from and thus may be a subject to change. Also, determining the attributes that will be stored by our application is a significant part of the design process and therefore will be further discussed in the Chapter 2.

The domain model of our application can be seen in the figure 1.2. There are multiple classes and relationships in the domain model that might benefit from further explanation.

1.3.1 Subscription

Subscription essentially represents a "watch dog", that sends the user a message to the specified channel once some change related to the specified region and district occurs. The subscriptions are not bound to particular addresses but rather to regions and districts. This is mainly due to the fact, that monitoring an address would be too specific while also being hard and costly to implement. Even though some users actually might want to only monitor a specific address, this is not the primary purpose of the application. It, however, definitely can be considered as a feature some time in the future if there is a demand for it.

1.3.2 RealEstate

RealEstate represents a listing of some real estate such as house or apartment. Our main use of the data will be sending notifications to subscribed users. This means that the scraped data from all the portals must conform to one

structure, as we want the notifications look uniform for all portals. Therefore, *RealEstate* model will contain only the portal non-specific information, that we can expect to be present in all listings. Examples of such information may be the type of estate, area of the land, price, url of listing etc. There are many more information present in the individual real estate portals, however these are of no use to us at the given moment. Therefore, such data will no be included in the database and as such has no place in the domain model also.

1.3.3 Address

Even though the class *Address* is rather self-explanatory, its relationship with *RealEstate* is slightly more complicated than it might seem. As the domain model states, each instance of *RealEstate* has exactly one *Address* instance related to it and one *Address* might have zero or more *RealEstate* instances related to it. This might sound like an error as in real world, it is not possible to have two properties with identical addresses. In our case, however, there are multiple good reasons for it.

1.3.3.1 Hidden location

When posting the real estate listing, its creator may choose to hide the exact location of the real estate. If this is the case, some portals will provide us with GPS coordinates that do not point to the specific real estate, but rather to some location in its proximity. This usually is the main square of the city or its town hall.

Imagine that we have two real estates, A and B. Let's also say that the following statements are true:

- The exact location of A and B was hidden by the creators of the listings
- In real world, A and B are located in the same street

In the case described above, the location that the portal is going to give us will be identical for both of these real estates as their real location is hidden and they are in proximity of each other. This means that we now have two real estates with the same address in our system.

1.3.3.2 Old listings

When sellers list a real estate for sale they might do it multiple times, even on one portal. The main reason for doing so is, that relisting the real estate will "bump" it to the top of all listings and make it more visible to the potential buyers. The sellers tend to do this either on periodic basis or when, for example, changing the price. In some cases, however, they also forget to remove the old listing prior to adding the new one. In this situation, our application will see two active listings that point to the same address and will

have the need to store them both. Therefore, limiting the maximum number of real estates per address would cause errors in this case.

1.3.3.3 Same real estate, different portals

In this case, we have one real estate listed on multiple different portals. As specified by our domain model, one real estate can be, in theory, linked to more than one portal's specific details. The process of identifying one real estate across multiple portals might be quite difficult, especially considering the issue described in previous subsection 1.3.3.1.

Even if we consider the implementation of the process above to be successful, we still expect it to not be perfect as matching all the listings perfectly, 100% of the time, is more or less an impossible task, even for some more advanced machine learning models which could be used to tackle this task. It seems to be much more sensible to not add unnecessary complexity to the application and rather slightly relax the relations in the domain model as the performance impact caused by this change most probably will not be noticeable.

1.3.4 Channel

When user sends a command to the bot, it always has an origin to which the bot's response is going to be sent. This origin is in Discord's realm called a *channel*. Essentially, channel is a chat in which conversation among two or more users takes place. Storing some information about the channel, such as its ID, from which the command that created the subscription originated, is therefore crucial to us as without it, we would not know where to send the response.

To further elaborate on Discord's architecture, there are also servers, which are basically groups with many users. Inside of these servers, multiple channels through which the users can communicate may exist.

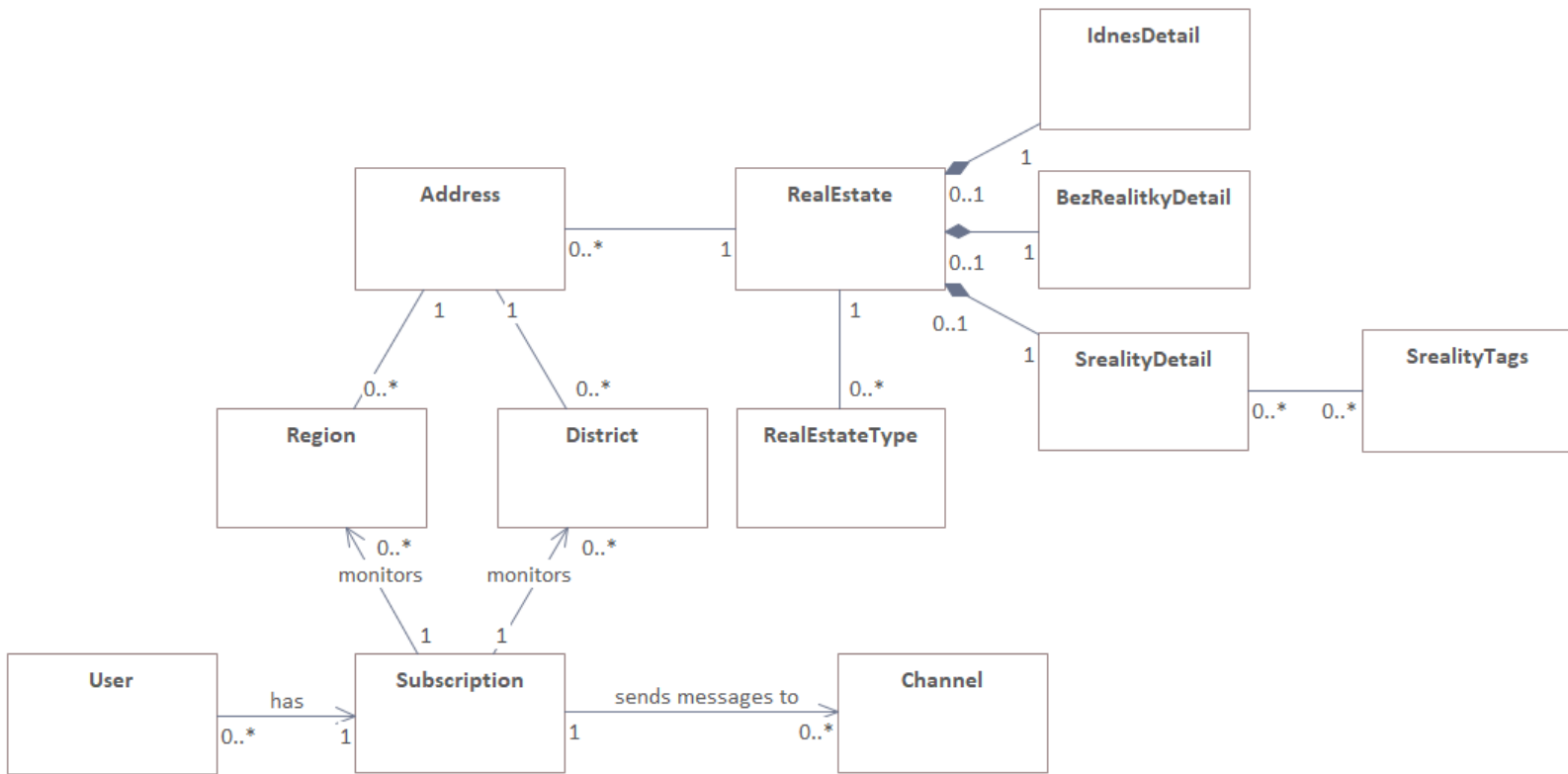


Figure 1.2: Domain model

Design

2.1 Database

As we need a place to store the data, running a database is necessary. The database will be used to store not only the data scraped from web portals, but also information on users, channels, subscriptions and more.

2.1.1 Data model

The data model / database scheme shown in the figure 2.1 depicts the relationships between different entities in our database. Each entity is represented by a standalone table in our database. It also shows the different attributes of different entities which translate to table columns in our database.

As all the relationships that are not intuitive or are in some way interesting were discussed in the section 1.3 dedicated to domain model, only some of the entities' attributes will be discussed here.

2.1.2 Real estate listings

In order to develop a data model that describes the data that the application is working with, we first need to analyze the data provided to us by the different web portals that are going to be scraped. This will help us determine which data related to real estates we can consider common between the portals and thus place them into the *RealEstate* table. The original thought was that the rest of the data will then be deemed web portal specific and will be placed into the table containing respective portal's details (*SrealityDetail*, *BezrealitkyDetail*, *RealityIdnesDetail*). While designing the data model, however, we realized that having the standalone tables for portal specific details does not make much sense. To elaborate, there are two main reasons for persisting the portal specific data, the first one being that we would like to create statistics on the gathered data at some point in the future and we are of the opinion

that some of the data, that might not necessarily be important to our users, or at least are not important enough to be parts of the notifications received by the users, could play an important statistical role while analyzing the data. An example of such data might be the points of interest, eg. distance from nearest hospital, school, post office etc., whether the listing was in any way promoted on the real estate portal or which agency/agent is responsible for offering of the given real estate. As there is many data of this nature present in the responses, it is essentially impossible to prioritize between them at this point in time. Therefore, creating a large data structure simply to store the data without actually knowing whether they will be useful is, in our opinion, an unwise investment of our resources.

The second reason as to why we want to persist said data is the fact, that purging potentially valuable and definitely interesting data seems like a waste.

Because of the reasons above, we decided to use AWS's service called Simple Storage Service, or as we will be calling it from now on, S3. S3 allows us to store objects such as .json or .xml files at a reasonable cost of 0.023\$ per GB/month [13]. This will allow us to store all the data that we request from the real estate portals for a low cost and parse them only once we are certain that we have a use of them, thus ridding us of the uncertain investment of resources at this point in time.

A great "side-effect" that this approach creates is, that if any serious issues with database happen and we end up losing the data, or we perhaps realize that there are some important data in the answers that we managed to miss during the scraping (especially the initial one), we can simply re-parse the answers from the files saved in the S3 and avoid scrapping the real estate portals again.

Going back to the creation of data model, some common attributes of real estates are expected to be part of the *RealEstate* table even before starting analysis of the data provided by portals. These are namely:

- title
- price
- location
- real estate type (house/apartment)
- land dimensions
- real estate dimensions

The attributes mentioned above are also expected to be part of all the notifications related to real estates that the users will receive. However, there are many important attributes that might not be part of the notifications, but

will be used for filtering purposes when creating subscriptions to provide the users only with notifications, that are relevant to their preferences.

After taking multiple sample responses from the scrapped real estate portals, we arrived at a list of attributes that are relevant for either notifications, or filtering, or both. As the list is quite exhaustive and the attribute names are quite self-explanatory, we will not list them all in this text as we did with the expected attributes and will rather refer to the figure 2.1 displaying the data model itself.

2.1.2.1 Address

The names of different attributes of the address may feel slightly odd or counter intuitive. As we are using Nominatim for reverse geocoding, it made sense to structure the table according to the output format of Nominatim's response to our geocoding request 2.1.

2.1.2.2 Suburb Prague

The table *suburb_prague* contains the mapping of Prague's suburbs to its districts 2.1. Why this table was created and how it is used is described in greater detail in the section 3.3.7.

2.1.3 Database type

There are multiple types of databases that we can pick from when choosing the database that we will use. We are deciding between the following ones:

- SQL
- NoSQL - Document
- NoSQL - Graph

The reason why we chose the three mentioned database types is mainly because SQL and document databases are the most frequently used types of databases [14]. Graph databases were chosen mainly due to personal preference as they seem like an interesting concept to us.

The data that we will be storing have a clear structure and follow it vast majority of time (all the time if we do not take errors and malformed data into the equation). There is also quite a lot of relationships present in the data structure and data related to users, channels and addresses are often repetitive. Due to these reasons, it does not make sense to use document database as it performs best when dealing with non-structured data and little relationships [15].

Both SQL and graph databases can be used for the data structure that our application will be working with. We decided to use an SQL database for multiple reasons, the first one being popularity [14].

2. DESIGN

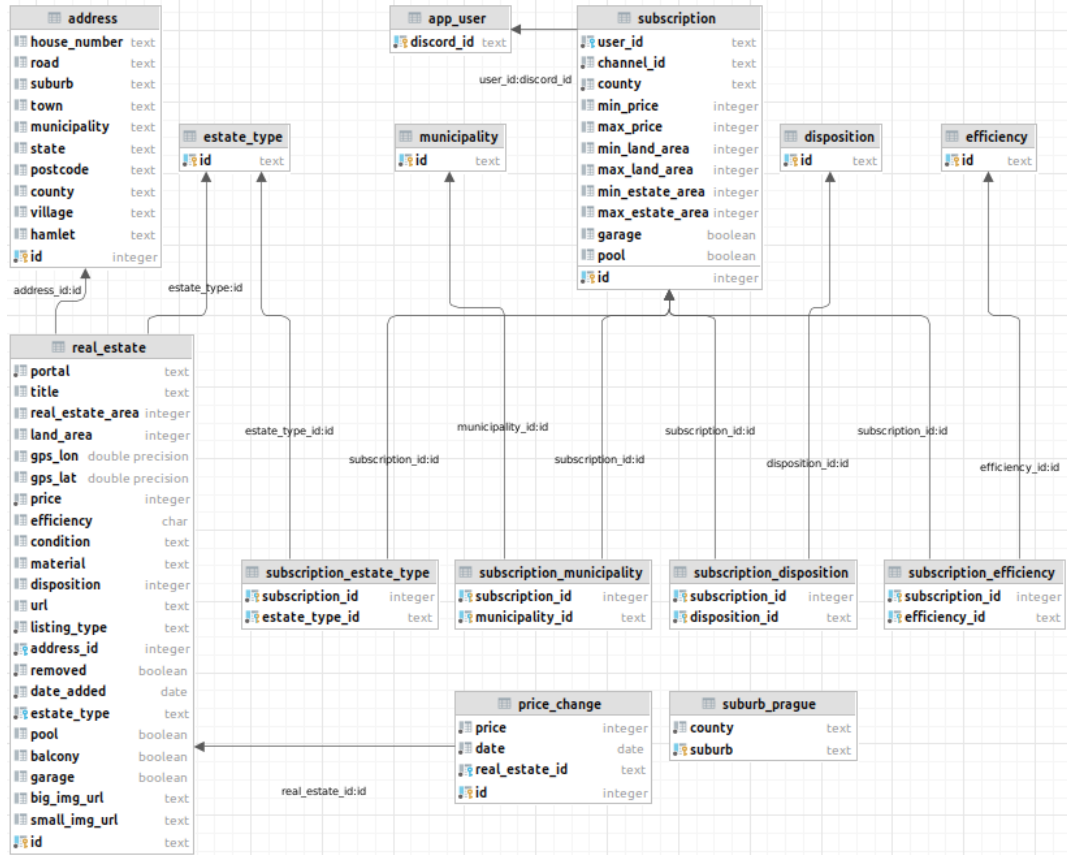


Figure 2.1: Data model

As SQL databases are much more popular and widely adopted than graph databases, there is a lot of content, libraries, support etc. regarding them. This makes it much easier to address any issues that occur throughout the development [14].

Another reason for choosing SQL database is the price of running it in the cloud. AWS provides a service called AWS Neptune, which is a fully managed graphed database. Its cost starts at 68 dollars per month due to the smallest supported instance being the ml.t3.medium which is unnecessarily powerful for our use case. Considering the fact, that the price of PostgreSQL managed instance starts at 12 dollars per month which is significantly less than the price of AWS Neptune, using an SQL database such as PostgreSQL is cost-wise much more sensible for our use [16].

2.1.4 RDBMS

There are many relational database management systems available. Some of the most used ones are Oracle, MySQL, PostgreSQL and Microsoft SQL Server [14]. All of these systems could without any issues be used to store the data of our application. We decided to use PostgreSQL as it is a RDBMS that we are familiar with from previous projects, is supported by all major cloud providers (AWS, Azure, GCP) and tends to be one of the cheaper options [17][18][19].

2.2 Scrapers

2.2.1 Programming language

There are many languages that can be used for web scraping. Ideally, we want to use a language that is very popular for this use case, because if we run into any problems it will be easier to find a solution to them. Also, a language that either has a shallow learning curve or a language that we already have an experience with is preferred. After doing some research, we will focus on the following languages as they seem to be the most popular ones for given purpose:

- Python
- JavaScript (Node.js)
- Ruby
- C/C++

2.2.1.1 Python

According to the TIOBE index [20], Python is currently the most used language in the world, thus checking our popularity requirement. According to multiple resources, it also appears to be one of the most popular languages for web scraping.

The first reason for that is its overall simplicity and very shallow learning curve as Python is often quoted as the most beginner friendly language [21]. Even though we already have experience with Python, having a simple language for web scraping is certainly taken as an advantage, because it will allow us to focus more on "what we want to do" instead of focusing on technical aspects of the language itself.

The fact that Python is a popular language for web scraping is noticeable just considering the amount of web scraping related packages that exist. Some of the most important packages are:

- requests

- beautiful soup
- Selenium
- lxml

These packages help with sending requests, parsing the received data, traversing it and much more and make the process of web scraping easier.

Python also has its challenges, with the most significant one being its speed. This is not much of an issue in our case, as there will have to be pauses between the individual requests that will be made so the scraped web portals are not saturated by our requests. Given the fact that the planned delay between requests is ~ 3 seconds, that gives Python more than enough time to perform all the operations needed.

One more thing to note, is that since we are implementing the scrapers in the form of a Discord bot, as discussed in subsection 2.2.2, we can use the discord.py library which, compared to its JavaScript counterpart, has a feature called task loops that allows us to easily set up periodic executions of chosen methods [22][23][24][25].

2.2.1.2 JavaScript (Node.js)

JavaScript itself may not be ideal for web scraping, however this is solved by Node.js runtime environment. This is due to the fact that JavaScript's original purpose was to be used for client-side development. Node.js changes that and allows developers to write server-side code using JavaScript.

One of the greatest advantages of Node.js is its rich ecosystem, that offers many third-party libraries and modules. Combined with JavaScript being a popular language for scraping, Node.js's ecosystem offers many modules and libraries that can be used while scraping and will save us a lot of time. Some of these modules are namely [26]:

- Axios - working with requests
- Cheerio - web page traversing
- Puppeteer - automating browser actions
- Playwright - automating browser actions

Another plus of using Node.js is the fact that it is backed by a strong community and there are lots of content regarding all the issues one can come across when working with it. This makes the whole experience of writing and fixing the code easier and faster.

Even though JavaScript paired with Node.js has many great features there are also some cons, the first one being that Node.js has a very limited standard library. That is an issue as we are forced to use third-party libraries for the

majority of operations that we need to perform. This introduces a greater risk of ending up with and unreliable code. We must also be careful when picking the libraries so they do not bring any vulnerable or in any way defected code into our projects.

Another weak point of Node.js is that the API's are callback based. This can lead to code which is hard to understand and unnecessarily complex, especially when nesting multiple callbacks. This can introduce a so-called *callback hell*, which is further described here [27].

One issue with Node.js is that the learning curve can be rather steep even when the developers have experience with JavaScript. That is probably going to be the case for us as we do not have much experience working with JavaScript [22][23][24][25].

2.2.1.3 Ruby

Out of the selected languages, Ruby has the disadvantage of being the least used one according to the TIOBE index [20]. This results in much smaller community and support compared to languages such as Python or JavaScript which in turn can make it harder to solve issues that we encounter during development.

Another issue when working with Ruby is its high memory consumption. This might be an issue because we try to use as small server instances as possible in order to make our application cost efficient [28].

Even though Ruby is often described as a language that is easy to learn, the syntax contains some specific and inconsistent "quirks". This sometimes makes it harder to write clean and maintainable code, especially for developers that are starting out with the language such as us [29].

One great advantage of using Ruby lies in the Nokogiri library, which is often used when scraping. The reason why Nokogiri is so popular is that it is able to deal with and parse broken HTML code. From a personal experience, when scraping web pages, the extracted code can often present itself as slightly broken, at least in the eyes of the scraping libraries. Therefore, having a library that is able to deal with such code is a huge plus and can save a lot of data from being discarded [22][23][24][25].

2.2.1.4 C/C++

C/C++ is a language that also came out as one of the most used languages for web scraping. Its main strength is performance, as C/C++ is significantly faster than Python, JavaScript and Ruby, however as mentioned in 2.2.1.1, speed is not of much importance in our case. Other than that, C/C++ is a complex language with a very low level approach and absence of proper web scraping libraries. Due to the stated reasons, we deem it not suitable for implementation of scraping bot.

2.2.1.5 Conclusion

In this section, we discussed multiple programming languages that can be potentially used for web scraping. The language that was discarded right away was C/C++ as using it for implementation of scrappers would be too complex and not necessary in our case.

Another language that was eliminated was Ruby. Even though Ruby is a great language for web scraping and the Nokogiri library is very powerful, picking it instead of Python or JavaScript would not make sense in our case, mainly due to its lower popularity and therefore smaller amount of resources and support.

Both Python and JavaScript have their pros and cons and can be used to implement scrapers comfortably and in our eyes are equally suitable for implementing web scrapers. In case of our scrappers we decided to go with Python due to multiple reasons.

First of all, Python is a language that we have a lot of experience with and are comfortable working with. This is quite the opposite to JavaScript and Node.js as we have very little experience with these technologies and the learning curve would therefore be quite steep for us.

Another reason for choosing Python is, that we decided to implement the scrapers as a Discord bot, as discussed in subsection 2.2.2. In order to do so, we need to be able to implement an equivalent of Cron jobs in the Discord bot, in other words, we need to be able to implement tasks that run periodically. Implementing Discord bot is possible in both Python and JavaScript as both of these languages also have Discord libraries, discord.py and discord.js respectively. These libraries differ quite a bit and some features implemented in one of them do not necessarily have to be implemented in the other one. That is exactly the case of task loops, which exist in discord.py and do **not** exist in discord.js. Task loops are an essential feature, because they allow us to implement periodically repeating tasks in a very elegant manner, which is exactly what we need for our scrapers.

2.2.2 Architecture

There are multiple options as to how the scrapers can be implemented. This subsection will discuss and compare the different options with the goal of choosing the most appropriate one for the purpose of our application.

2.2.2.1 Server-less/Lambda functions

The first way the scrapers may be implemented is as server-less functions. This way of implementing the scrapers is available to us thanks to using a cloud provider to host our application. The term *server-less* functions describes functions, that do not require some specific instance of server. This means, that when the functions are executed, the cloud provider lends us a server

from their pool of servers.

Costs

The fact that the server is being lent to us can be cost efficient, as we do not have to run our own instance, but we can just borrow an existing instance from the cloud provider's pool and be billed just for the execution time, typically in milliseconds, of our functions. This statement holds mainly in the scenarios where the functions are either called infrequently or are very short. In case of our application, the functions that scrap new listings are called roughly every 5 - 15 minutes and the functions that look for deleted listings and price changes run twice a day. The estimated time for each execution of the functions that scrap new listings is 30 seconds because there will only be a few new listings during each run as the scrapers run quite frequently. On the other hand, the functions used to scrap removed and changed listings need to go through all the existing listings. Expecting to have roughly 50,000 listings at 60 listings per page, this equals to ~ 850 requests each time the function is ran. Considering the fact that we want to wait 3 seconds between requests on average, this gives us the total execution time of ~ 45 minutes. The sum of all executions during one day totals out to roughly 14,000 seconds. Calculating this for AWS Lambda functions gives us a total monthly cost of ~ 7 dollars, which is slightly more than running an equivalent AWS EC2 t4g.micro instance at ~ 6 dollars per month.

Configuration

Another feature of server-less functions is that they can be easily configured to run periodically, thanks to the serverless framework. Serverless framework is a solution allowing us to easily deploy and schedule server-less functions. It supports many of the largest cloud-providers [30] such as Amazon AWS, Microsoft Azure or Google Cloud [31]. Also, serverless supports multiple languages including Python, which we are using to implement the scrapers [32].

As we will want to scrap the data on a periodic basis, the ease of configuring the functions is definitely important to us.

Persistent storage

One of the weak points of server-less functions is the inability of storing data directly on the servers. As the instance of server on which the server-less function runs changes each times, it is not possible to store any data there. Therefore, most cloud providers provide a solution that allows the server-less functions to persistently store data. Using these solutions, however, can be more complicated as setting up the storage and connecting it to the server-less functions takes more time than directly working with file system. An example of this can be using AWS Lambda functions with AWS EFS [33]. On the other hand, some providers such as Microsoft Azure, run the server-less functions

with a persistent storage attached by default [34].

Availability

Reason why server-less functions may not be the ideal solution for us, is that the application is available only when it is being run. This would make the process of manually starting the scrapers 1.2.1.9 more complicated as instead of just starting the scraping process inside the application, we would have to manually start the whole server-less function. This means that we would have to deal with authentication towards the account that we would have with the chosen cloud provider and also would not be able to start the scrapers directly from the Discord application, which would be preferred.

Internet access

One issue that comes with using server-less functions is when trying to access the internet. As we want to keep our application safe, we plan to have all its parts enclosed in a private network. This creates an issue, as when we add server-less functions to a private network, we have to create a NAT Gateway, which is used to route the outbound traffic from server-less functions to the internet. Running a NAT Gateway is, however, quite expensive, as it requires a standalone instance of virtual server, which increases the cost of running the application significantly [35][36][37]. If we take a look at AWS, the cost of running NAT Gateway is ~ 32 dollars per month, which is much more compared to running a t4g.micro instance.

Time limit

Server-less functions solutions typically have a limit that prohibits the functions from executing for long time. In case of AWS and Azure, these time limits are 15 minutes and 10 minutes respectively [38][39]. This is an issue, as some of our scrapers will most certainly exceed the 10/15 minute limit. This issue can be tackled by making the scraping functions more granular and calling them with parameters, specifying exactly which smaller part of the web portal are the functions supposed to scrap. That would allow us to scrap the web portals using timewise shorter executions of the functions and therefore not exceeding the time limit.

The issue that may come up with this approach is that scheduling the functions will become more complicated, as we cannot allow multiple executions to run in parallel because it could cause an excessive load on the servers that they would be scraping.

It is worth noting that when using Azure Functions Premium plan, the execution time of function is unlimited. The premium plan, however, comes at a cost, which in our case exceeds the cost of running an instance of a standalone virtual server with equivalent hardware [40][41].

Scalability

One of the features that the server-less functions offer is great scalability. It is very easy to configure how much resources can each function use as well as how many instances can run in parallel. The scalability is perfect for use cases such as APIs, where the usage of the functions may be unbalanced and the number of calls may spike throughout the day as there is no need to deploy new servers and wait for them to load and therefore the scaling happens instantly.

In our case, the time complexity of the scrapers is and will remain more or less constant, as the amount of scraped data is not dependent on the number of users that the application has. Therefore, we do not expect the scrapers to require scaling.

2.2.2.2 Cron jobs

Another option as to how the scrapers can be implemented are Cron jobs. Cron job is a term for a job created by a scheduling tool called Cron that is present in Unix-like operation systems. Cron jobs can be configured to run at a specific time, date or frequency, which is perfect for our use [42].

In case we were to implement the scrapers as Cron jobs, each of them would be implemented as a standalone application and would be configured to run periodically using the Cron tool.

Configuration

As we plan to use Docker to containerize all parts of our application to simplify the deployment, we can simply define the Cron jobs in the Docker file of the respective Docker container inside which the scrapers will be running [43].

Costs

The scrapers do not require much hardware resources and their complexity does not scale alongside with the number of users and stays more or less constant throughout the time. Therefore, it is expected that instances with 1GB of RAM, and 2 vCPU cores will be sufficient in the worst case scenario. If we were to use AWS as our cloud provider, the cost of running such instance of a server would be ~6 dollars per month.

Availability

Availability of the scrapers might be an issue with this approach. As with server-less functions, the application could be communicated with only when it is running, which would be only during the times specified in Cron. Manually starting the applications would be quite complicated as it would be needed to allow an ingress to the instance of server that the scrapers are running on and from it to our Docker container. Therefore, we would always have to know the IP address of the server when communicating with scrapers. Also, in order to

start the scrapers in the Docker container, we would have to create an API, that would be able to start the scrapers on command which would add further complexity to our application.

2.2.2.3 Discord bot

At this point there are two options of implementing the scrapers, server-less functions and Cron jobs. Neither of these options seem to be ideal in our case.

The server-less functions can be easily configured and manually triggering them can be implemented with reasonable additional complexity. Unfortunately, when running the functions inside a VPC, NAT Gateway is required, which in case of AWS costs ~ 32 dollars per month.

The Cron jobs have a low cost of running and can be easily configured when using Docker [43], however manually triggering them is not easily doable and would require the development of another API.

At this point, we are looking for a solution that would allow us to manually trigger the scrapers without the need for an API whilst remaining cost efficient and easily configurable.

A potential solution for this would be another Discord bot.

Configuration

As we are using Python to implement the scrapers, we can also make use of the discord.py package. One of the things that discord.py package allows its users to implement are task loops. Task loops are basically an equivalent of Cron jobs as we know them in Unix-like systems, allowing us to run some method at specified time, date or frequency. Thanks to this feature, it is simple to configure the scraping functions to run when needed [44].

Costs

This Discord bot will also run in a Docker container, allowing us to simply deploy it to an instance of virtual server. As previously stated in subsection 2.2.2.2, the cost of server, matching the hardware requirements of the scrapers, is estimated at ~ 6 dollars per month which is acceptable.

Availability

As the scrapers are implemented as task loops in Discord bot, it is possible to add commands to the bot, which, when called, execute the scraping function. This approach is very simple as it does not require implementation of another API (Cron jobs) nor access to the cloud platform itself (server-less functions). Instead, the scrapers can be communicated with through a Discord chat and are available at all times.

2.2.2.4 Conclusion

In this subsection, three different options as to how the scrapers can be implemented were discussed. From these options, Discord bot came out as a clear winner.

Its main advantage compared to both server-less functions and Cron jobs, is the ease of implementing the manual execution of the scrapers and overall the communication with them such as health checks.

One thing that was not discussed earlier is the "client side" when communicating with scrapers. If we were to implement the scrapers as server-less functions or Cron jobs, we would also have to implement some simple web application that we would use to communicate the scrapers. This would also require implementing some form of authorization as we would not want other people to manually trigger the scrapers as discussed in 1.2.2.5. When implementing the scrapers as Discord bot, all these responsibilities are taken care of by the Discord application as it will be used for communication.

Also, compared to server-less functions, the costs of running a Discord bot are significantly lower while the ease of configuration remains.

2.2.3 Commands

As the scrapers will be implemented in the form of Discord Bot, it allows us to create commands to communicate with them. These commands as well as the bot itself will be only available to the administrator of the application, as this part of the application is not meant to be client facing.

In case of this bot, we will implement only two commands.

The first command will have the responsibility of checking the health of the Discord bot, eg. if it is running. The command will be in the form ping and when called will simply respond with text, indicating that it is working.

Another command that we implemented is in the following format: *!scrap* *{type}* *{portal}*. The *type* allows three different values:

- price - will check price changes
- new - will check newly added listings
- removed - will check removed listings

The *portal* argument is used to specify the targeted portal. The allowed values are:

- sreality
- idnes
- bezrealitky

Whilst running the bot in a development environment and working with it, we realised that we are able to check for changes every 5 minutes or even more often as the load imposed on the scraped servers is minimal. This rendered the command obsolete, as the scraping frequency is so high that there is no possibility of our data being outdated and needing a manual refresh. This command was therefore removed from the bot during one of the development iterations and is no longer available.

2.3 Discord bot - client side

This Discord bot will be the UI of our application. It will allow the users to create, update and remove their subscriptions as well as inform them about listing changes that they are interested in and more.

In this section, we will discuss the commands that can be used to communicate with the Discord bot and the choice of programming language that the Discord bot is going to be implemented in.

2.3.1 Commands

The only way the users can communicate with bot is through commands. Commands are created the same way any other message in a channel is created, however they also have to follow a specific format. In our case, each message that is preceded with a backslash (\) is considered a slash command by the Discord bot. Slash commands differ from standard commands in the way they communicate with the user. When using standard commands, we cannot really interact with user. In case of slash commands, we can edit, delete and create messages and follow-up on them.

In the following text, *text in italics* signifies a command. Each command will also be preceded, as noted earlier, with a backslash (\). An example of command might look like this: `\command`

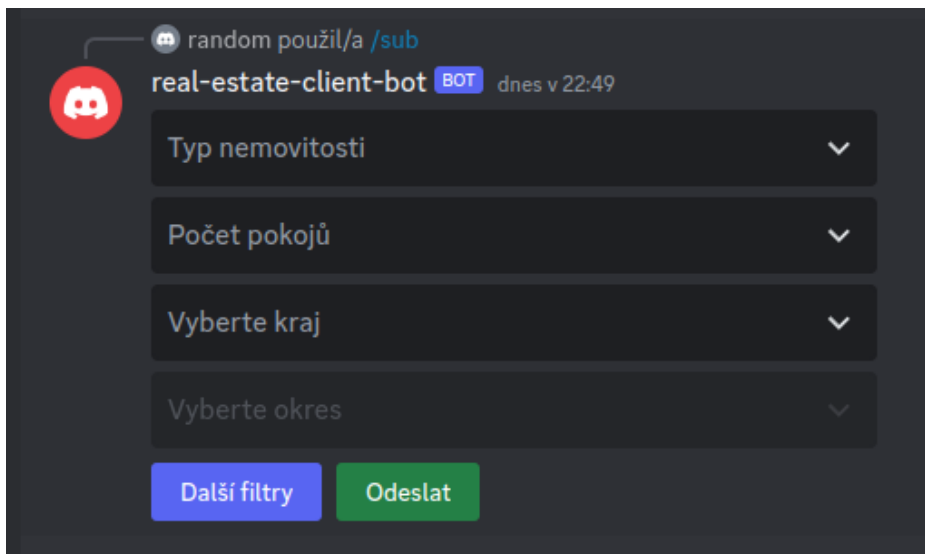
Based on the functional requirements, we need to design commands that will allow the users perform all the described actions. These actions are namely:

- Create subscription
- Create subscription with filters
- Update subscription
- Delete subscription
- Show subscription details
- Check health status

In general, we want the commands to be short, easy to remember and as intuitive and self explanatory as possible. This will help the users when learning to interact with the Discord bot and therefore can have a significant impact on the success of the whole application.

2.3.1.1 Creating a subscription

Creating a subscription with and without filters share the same command in the form of: `\sub`. After sending in this command, a response containing a form will appear. This form is prompting the user to fill in the basic subscription parameters, as can be seen in the figure 2.2. It also contains two buttons, one is for submitting the form with the filled data, the other one is to show more parameters, voluntary and more detailed ones, that belong to the subscription. This form can be seen in the figure 2.3. After filling some more details, the user can submit the filled in form using the submit button. This will conclude the process of creating a subscription and will print out and embed containing information about the newly created subscription. The embed can be seen in the following figure 2.4. The forms used to the create the subscription will be deleted. The whole form with all the parameters filled out can be seen in the figure 2.5.

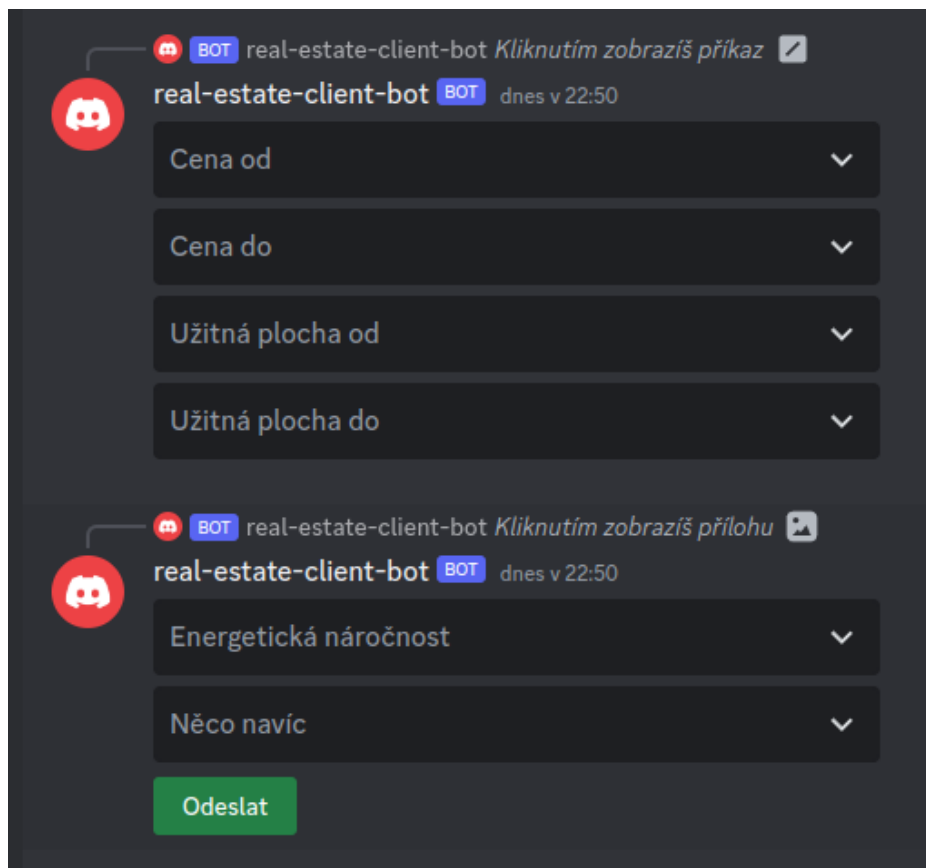


The screenshot shows a Discord chat window. At the top, it says 'random použil/a /sub'. Below that, the bot's name 'real-estate-client-bot BOT' is displayed with a timestamp 'dnes v 22:49'. The main content is a form with four dropdown menus: 'Typ nemovitosti', 'Počet pokojů', 'Vyberte kraj', and 'Vyberte okres'. At the bottom of the form, there are two buttons: 'Další filtry' (blue) and 'Odeslat' (green).

Figure 2.2: Client bot - create form - basic

2.3.1.2 Updating a subscription

Updating a subscription is done through the following command `\sub_update`. As a response to this command a select menu containing all the user's sub-



The image shows a Discord chat interface with a dark theme. At the top, a bot named 'real-estate-client-bot' is shown with a red 'BOT' label and a 'Kliknutím zobrazíš příkaz' link. Below this, the bot sends a message with a red Discord icon and a 'BOT' label, dated 'dnes v 22:50'. The message contains four dark grey dropdown menus with white text and downward arrows: 'Cena od', 'Cena do', 'Užitná plocha od', and 'Užitná plocha do'. Below these, the bot sends another message with a red Discord icon and a 'BOT' label, also dated 'dnes v 22:50'. This message contains two dark grey dropdown menus with white text and downward arrows: 'Energetická náročnost' and 'Něco navíc'. At the bottom of the second message, there is a green button with white text that says 'Odeslat'.

Figure 2.3: Client bot - create form - filters

scriptions is returned. This response can be seen in the figure 2.6. After picking an option from the select menu, the same process and therefore the same responses take place as when creating a subscription. The only difference is that the form data are pre-filled with the data of the selected subscription.

2.3.1.3 Deleting a subscription

Deleting a subscription is done through the following command `\sub_delete`. As a response to this command a select menu containing all the user's subscriptions and a confirmation button are returned. After picking a subscription to delete from the select menu, the user confirms his choice by clicking the confirmation button. After that a simple text message confirming the operation is returned.

2.3.1.4 Viewing subscription details

Viewing subscriptions details is done using the following command `\sub_detail`. As a response to this command a select menu containing all the user's subscriptions is returned. After picking a subscription to view from the select menu, the bot will respond with an embed containing the details of the selected subscription. The embed can be seen in the figure 2.4.

2.3.1.5 Health-check

In order to check the health of the Discord bot, eg. if it is running, the users can use command: `\ping`. This command returns simple text message, just to indicate that it is running.

When using commands of the client bot, the users also get an overview of all the existing commands after typing in `/` into the chat. This overview displays the names through which the commands may be called as well as their descriptions. The overview can be seen in the figure 2.7.

2.3.2 Programming language

Discord bots can be implemented using multiple languages. These are namely:

- Python
- JavaScript
- Java
- C/C++

2.3.2.1 C/C++

Even though implementing a Discord bot using C/C++ is possible it tends to be quite complicated and is not done very often as the vast majority of developers picks either Python or JavaScript for their implementations [45].

2.3.2.2 Java

It is possible to implement the bot in Java and some packages such as JavaCord or Discord4J that simplify the development are available. Despite that, the majority of Discord bots are nowadays implemented using Python or JavaScript. Because of that, there is not as much resources available regarding development of Discord bots in Java as there is for development in Python or JavaScript. That is why for our Discord bot, we will also be deciding mainly between Python and JavaScript.

2.3.2.3 Conclusion

JavaScript and Python are frequently used to develop Discord bots and feature their own libraries that make the development significantly easier, namely `discord.py` and `discord.js`. Both of these languages seem to be a great choice, with Python being seemingly slightly more popular for this purpose and therefore the choice comes to personal preference.

For our client-side Discord bot we decided to use Python. The main reason for doing so is that we will already be implementing the scrappers as Discord bot in Python and picking Python also for the client-side Discord bot will allow us to reuse parts of the code such as the model classes.

Also, we do not have much experience with JavaScript which could make implementing the Discord bot quite challenging. By using Python, we eliminate the need of learning and on-boarding more technologies to our application, thus keeping the application more simple and allowing us to reuse the knowledge gained by implementing the scrapers [46][47].

2.3.3 RESTful API

2.3.3.1 Programming language & framework

There are many different frameworks for various programming languages that are used for API development nowadays. The most used ones are namely [48][49][50]:

- Spring Boot
- Django REST
- Express.js
- Flask
- Ruby on Rails

All of these frameworks are popular, have a large community behind them, are actively maintained and in our opinion are all suitable for the development of our API. The choice therefore comes down to a personal preference. We decided to go with Spring Boot as we are used to it and have previous experience working with it. We are also more comfortable working with Java compared to Ruby, Python and JavaScript, as it is the only statically typed language of them and also features checked exceptions which, in our opinion, is a good functionality to have as it makes the code more reliable.

2.3.3.2 Endpoints

In our API, there will not be many endpoints that will support *GET* requests. This is due to the reason, that the data provided by the API will be

transmitted using AWS SQS. The main purpose of the endpoints will be the following:

- create and update real estate listings
- create, retrieve, update and delete subscriptions
- create new users
- create new addresses
- create new objects representing price changes

All the endpoints described below will return data in the JSON format. The supported MIME type will be application/json.

GET **/users/{user_id}/subscriptions** - returns all existing subscriptions of user specified by *user_id*

POST **/subscriptions** - creates new subscription

GET **/subscriptions/{subscription_id}** - returns the subscription specified by *subscription_id*

PUT **/subscriptions/{subscription_id}** - updates the subscription specified by *subscription_id*

DELETE **/subscriptions/{subscription_id}** - deletes the subscription specified by *subscription_id*

POST **/users** - creates new user

POST **/addresses** - creates new address

POST **/price_changes** - creates new price change object

POST **/real_estates** - creates new listing

PUT **/real_estates/{real_estate_id}** - update the real estate listing specified by *real_estate_id*

2.4 Cloud provider

One of the goals of this thesis is to deploy and run the application in a cloud environment. This will ensure that the application is available and can be easily scaled when needed. This means that a cloud provider, that is suitable for our application, has to be chosen.

Our priority when choosing the cloud provider is for it to be as frequently used in companies as possible, thus making the newly acquired skills reusable in the future. For this reason, we are considering these three cloud providers, as they are at the moment the most used ones in the world [30][51]:

- Amazon Web Services (AWS)
- Microsoft Azure

- Google Cloud Platform (GCP)

In the past, we have worked with all the aforementioned cloud providers and therefore already possess some experience regarding them. As our experience with GCP was not one of the best and we did not find using its UI satisfactory, the decision lies between AWS and MS Azure. Both of these providers offer similar services and we have good experience working with them. The provider that we slightly favor is AWS, as it features much larger market share compared to MS Azure, especially among the companies [30][51]. Another reason why we prefer AWS over MS Azure is because we have more experience working with it and we would also like to broaden the skills we already have.

2.5 High-level architecture

This section will describe the high-level application architecture. It will also briefly describe the relations between the different parts of the application. The description of the application parts / services will be provided in their respective sections.

Diagram depicting the original high-level architecture can be seen in the figures 2.8. This diagram represents both the architecture when scraping the data for the first time and when scraping it periodically. The only differences in those cases will be the location of the Nominatim server. That is because in the initial stage, when we will be scraping a large amount of data from the web portals, we would be bottle-necked by the Nominatim public API's limit of 1 request per second [52]. That is why we will at first deploy our own instance of Nominatim service locally and after the initial scraping of data switch to the public one.

However, during the development of data model in the section 2.1.1, we arrived at a conclusion, which slightly changes the overall architecture. The updated high-level architecture schema shown in the figure 2.9 contains an additional AWS S3 bucket. The reason as to why the S3 bucket was used is explained in the section 2.1.1

2.5.1 Deployment

We decided to containerize all the application using Docker. This will allow us to easily scale different parts of the applications independently whenever needed. The docker images of each part get built using a GitHub Action and are then pushed to AWS ECR. After that, we use AWS ECS to run the images. We decided to use AWS ECS as it is free, easy to configure and able to deploy containers to EC2 instances in selected auto-scaling groups.

2.5.2 Managing listings - Scrappers and REST API

When scrappers obtain and parse data, they need to persist the newly acquired information. This is done through communication with the REST API, namely its listings related endpoints. One could argue that the scrappers could access the database directly, however this could cause various issues. The main issue is, that as there might be some more complex integrity constraints that need to be met by the inserted data, the code that represents them would have to be reimplemented/reused throughout all the scrappers. By allowing only the instances of this API to perform write operations on the database, we only need to have the integrity constraints in one place.

2.5.3 Managing subscriptions - Discord bot and REST API

When users create new subscriptions, we need to persist them in our database. Due to very same reasons as mentioned above, we deem it suitable to write this data into the database through the subscription related endpoints of the REST API.

2.5.4 Obtaining user info - Discord bot and REST API

When the Discord bot receives messages from queue, it needs to distribute them among the users that are interested in them. In order to do so, it needs to obtain data related to subscriptions. This is also done through the REST API as it already writes subscription related data into the database.

2.5.5 Reverse geocoding - Nominatim and Scrappers

Nominatim is a service that allows us to perform reverse geocoding. In other words, it allows us to take GPS coordinates, pass the coordinates to Nominatim and in return get address of the location closest to the given GPS coordinates. As most of the real estate portals do not share the exact address of the real estate, but do share GPS coordinates, our scrappers are only able to gather GPS coordinates of listings. Nominatim allows us to convert this information into address, which is very convenient in our case as the user will be subscribing to regions or municipalities, not GPS coordinates.



Figure 2.4: Client bot - subscription embed

The image shows a screenshot of a Discord chat interface. At the top, a user named 'random' has used the command '/sub'. Below this, the bot 'real-estate-client-bot' (BOT) has responded with a series of dropdown menus. The first dropdown contains 'Dům' and 'Byt'. The second contains '2', '3', and '5'. The third contains 'Karlovarský kraj'. The fourth contains 'Okres Cheb', 'Okres Karlovy Vary', and 'Okres Sokolov'. Below this, the bot has sent a message with a checkmark icon: 'Kliknutím zobrazíš příkaz'. This is followed by another set of dropdown menus: 'Cena > 2,000,000 Kč', 'Cena < 3,000,000 Kč', 'Užitná plocha > 40m²', and 'Užitná plocha < 60m²'. Below this, the bot has sent a message with an image icon: 'Kliknutím zobrazíš přílohu'. This is followed by a final set of dropdown menus: 'B', 'C', and 'D', and 'Garáž'. At the bottom of the chat, there is a green button labeled 'Odeslat'.

Figure 2.5: Client bot - filled out forms

2. DESIGN

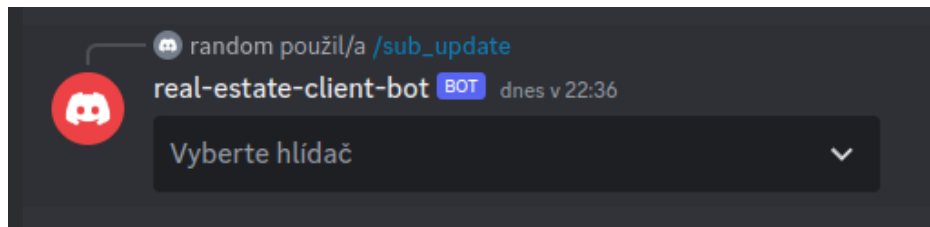


Figure 2.6: Client bot - subscription selection

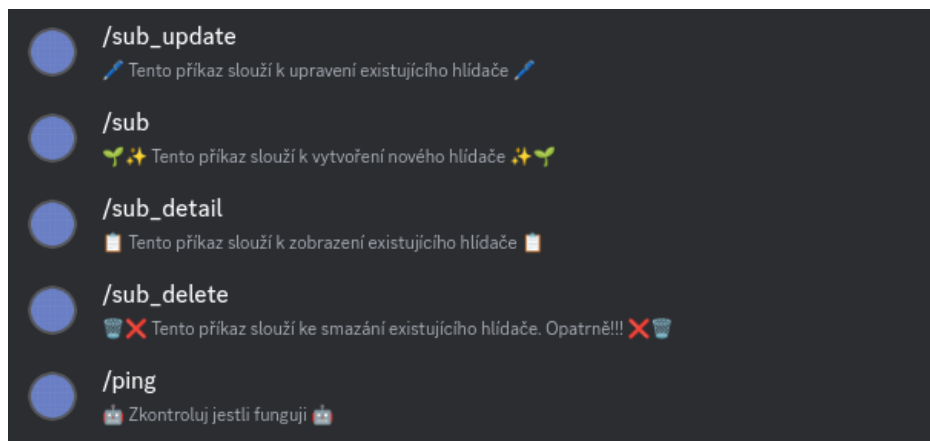


Figure 2.7: Client bot - command overview

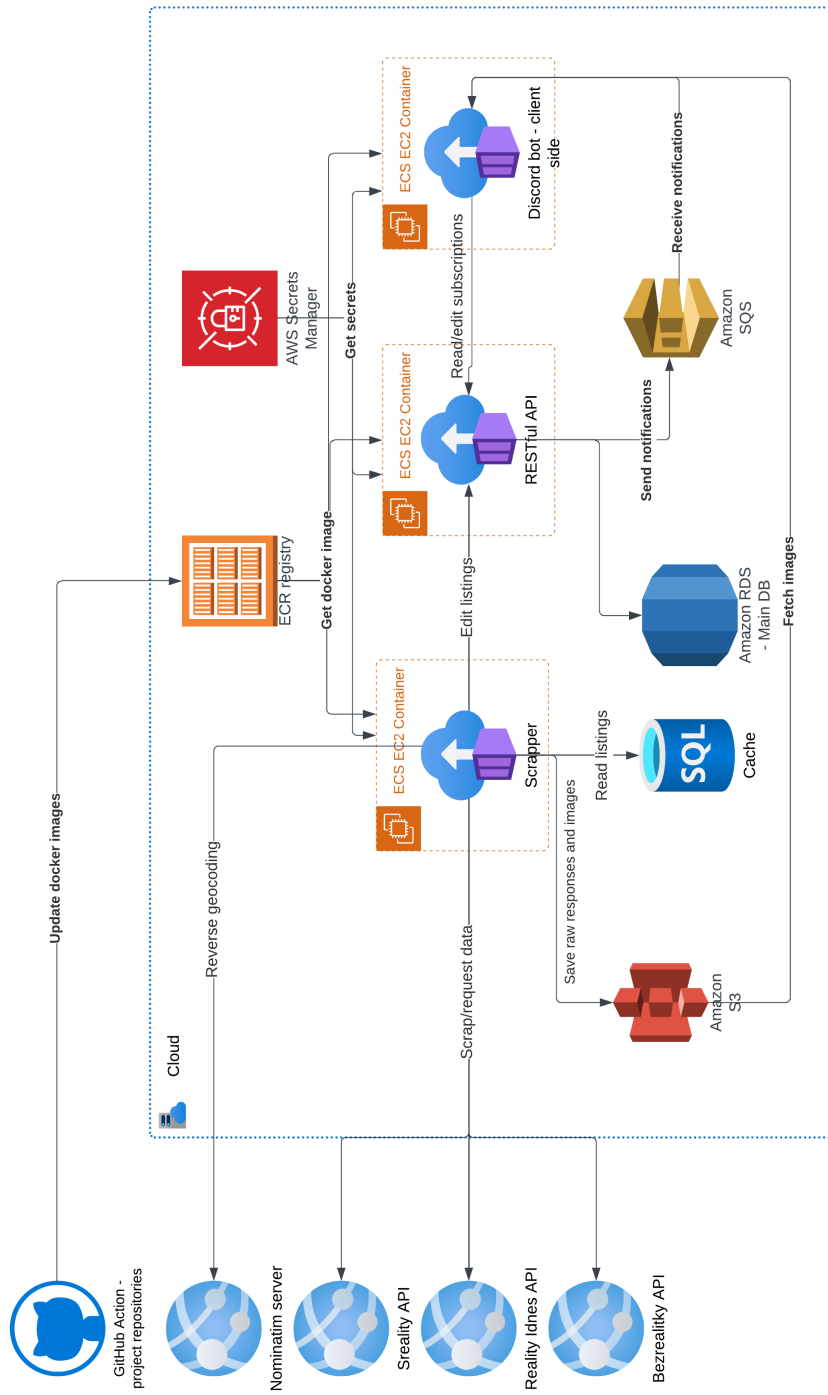


Figure 2.8: High-level architecture

Implementation

3.1 Scraping Bot

3.1.1 Sreality scrapping

One of the web portals that is going to be scraped is Sreality [53]. The two main categories that will be scraped are apartments and houses which totals out to roughly 40000 listings. There are two different approaches to scraping this data which both have their pros and cons

3.1.2 Scraping new listings

3.1.2.1 Scraping data from search results

With this approach only the data present in the search results would be scraped. As we can see in the figure 3.1 some important listing attributes are missing, namely:

- Energetic efficiency
- Layout of the property (number of rooms)
- State of the property

Also we can see that the dimensions of the property and the land as well as the type of the property are not explicitly stated anywhere, except for being part of the listing title. There is now way to be certain whether the listing title is machine generated or not, however based on manually reviewing a couple hundreds of results it will be considered to be machine generated. This implies that dimensions of land and property and the type of the property can be determined by parsing the title of listing.

There also is a way to obtain the energetic efficiency, layout of the property and property type without viewing the details of each of the listings. Sreality[53] has search options, which allow us to filter the listing by exactly the

attributes that we are missing. Therefore when, for example, searching for a house in *very good* condition, with 6 rooms and energetic efficiency rating of A , all the results will have exactly these values of the missing attributes and therefore we will know them.

The main benefit of this method is, that running the application and obtaining all the data for the first time would be fairly lightweight as it would not require more requests than the following scrapings of newly added listings.

This method also comes at a cost. In order to scrap new listings and get all the attributes that we are interested in, we have to try all the combinations of the values of the filters that we are interested in. As there are:

- 6 types of property layouts
- 10 options regarding the property state
- 7 options regarding the energetic efficiency of property

All combinations of these options total out to 420. This means that each and every time new listings would be scraped, 420 requests would have to be performed. Even though 420 requests is not that many, it certainly would prevent us from refreshing the data frequently as scraping the listings, for example, four times per hour would impose quite a big load to the Sreality API server of over 1000 requests.

3.1.2.2 Scraping data from listing details

This approach slightly differs from the approach described in the subsection 3.1.2.1. As the name suggests, the listings data are not extracted from the list of listings returned as a search result, however the search result still plays an important role as it is used to obtain the links to details of all listings. Once we get all the links to listings details we have to scrap them one by one.

The main advantage of this approach is that we will not be missing any attributes such as the previously mentioned energetic efficiency, property layout and state of the property as the details page contains all the attributes stored by Sreality. Therefore, when scrapping newly added listings, it is not necessary to try all the combinations of missing attribute values. This allows us to scrap/look for newly added listings more frequently as we will not put the load of hundreds of requests on the server every time, but we will only need one request per newly added listing plus one request per a page of listings (usually 60 listings per page).

The con of this approach is that the first time the application is ran, it will need to scrap all of the listings details, thus performing tens of thousands of requests. This is not much of a complication as this has to happen only once during the whole lifetime of the application and therefore the process of the initial scraping can be dragged across multiple days, thus lowering the load on the scrapped servers.

3.1.2.3 Conclusion

For the reasons mentioned in the subsections 3.1.2.1 and 3.1.2.2, we decided to scrap the listings both initially and periodically from the details of each individual listing. In general, these approaches were available not only for Sreality, but for all the real estate portals that we scraped. Therefore, the decision of scraping the data from listing's details and not from the page of listings holds for all the selected real estate portals.

3.1.3 Address scraping

At this point it might also seem like we are missing a crucial attribute of the property which is its address, as it can not be seen in the search results in figure 3.1. Even though the exact address is not visible in UI of the website, each of the Sreality API responses containing the listings also contain GPS coordinates for each of the listings. These coordinates then can be used to determine the address of the property. The process of obtaining the address from coordinates is called *reverse geocoding* and will be discussed in the section 2.5.5. Important thing to mention is that the creator of the listing may choose to hide the exact location of the property. That is not an issue as the defined use-cases do not require it. Also, even though the exact location is hidden, the GPS coordinates will still be present in the API response. In this case, the coordinates will be pointing to a random address which is in proximity to the listed property.

3.1.3.1 httpx vs requests

Initially we tried to use requests package to communicate with API of Sreality. After a while, it became apparent, that the data in the responses are dated, malformed and inconsistent. Through the method of trial and error, we arrived at the conclusion, that the used version of HTTP is to be blamed. From the sent requests by the requests package, it seems like the version HTTP 1.1 was used, however when using a newer package called httpx, the HTTP 2.0 version was used. For a reason unbeknownst to us, the responses started coming back without any error once the HTTP version 2.0 was used and trying to state a reason for that would be pure guessing and speculation. None the less, the used HTTP version affecting the data in responses is a rather interesting observation.

3.1.3.2 Reverse engineering of the API

As the API that we are working with has no documentation, we have to figure out a way to use it. In order to do so, a combination of developer tools, PyCharm IDE and Burp Suite were used. First, we logged the network traffic through the developer tools of the browser to get the gist of how the website

3. IMPLEMENTATION

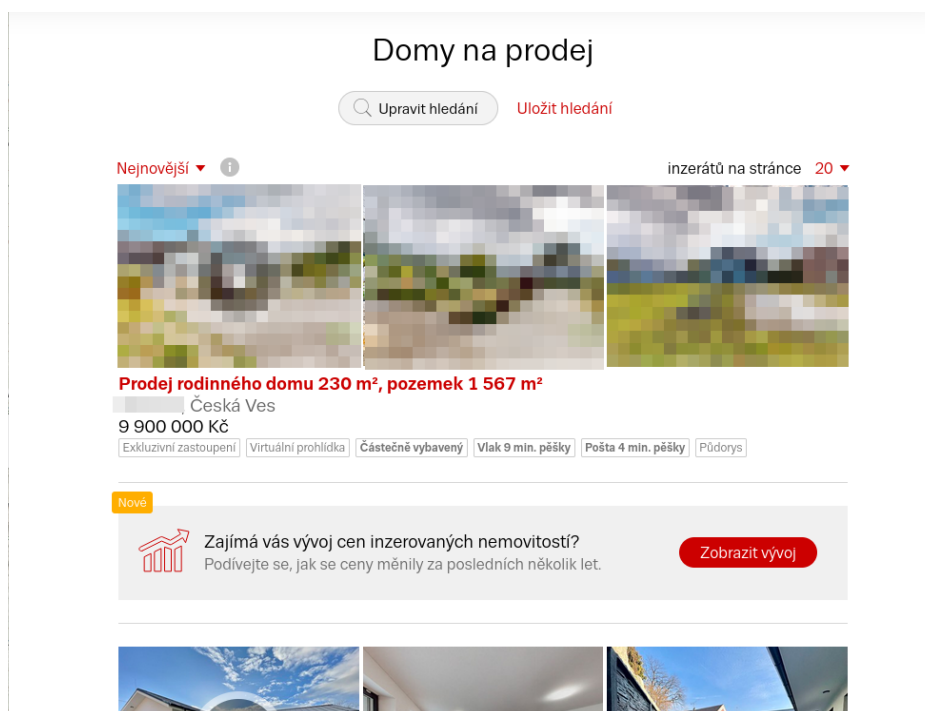


Figure 3.1: Sreality house search results

is behaving. After narrowing the scope and figuring out which requests are of importance we moved to Burp Suite in order to intercept the requests and learn more information about them, such as how to modify them in order to obtain the data we want. As the responses were often lengthy, we used PyCharm IDE to locate the elements that we were interested in and figure out how to extract them from the responses.

This process was more or less the same for all the real estate portals that were scraped. Therefore, it will not be repeated again in the sections devoted to the other real estate portals.

3.1.4 Reality Idnes scraping

3.1.4.1 Addresses

One of the main issues with the Reality Idnes portal is that the addresses have no standardized format in which they are presented on the website. Also, only part of the address tends to be visible to the website user and information such as the region in which the estate is located tends to be missing.

This is a problem as we rely on the location of real estates in our subscriptions as the users specify which localities they are interested in. Therefore, we have to figure out a way to obtain a complete address of each listing

The first option we tried was using OpenAI's model such as Ada, DaVinci or GPT-3.5. We sent the available part to the models and task them with completing given address. This was not a solution as the responses were often wrong, not following any structure and hard to work with in general.

After thoroughly examining the source code of the website we noticed that in every listings detail there is a small map showing a tag in the location of the given real estate. Looking at the code which was used to render this map onto the website, we noticed that GPS coordinates are being used to tell the website where to draw the tag. We were able to extract these coordinates and pass the to Nominatim instance for reverse geocoding which responded with perfectly formatted full address.

3.1.5 Bezrealitky scraping

3.1.5.1 Scraping listing's details

In order to scrap the details of individual listings, we wanted to send a request to the API of Bezrealitky asking for the respective linking and hoped to receive a JSON response which would be great to work with. However, When calling the API directly there is token that needs to be part of the URL for each request. After examining the websites behavior and mainly the requests it sends, we were able to see that the token is generated by a JavaScript function which was not very readable as it was obfuscated.

In order to get the token which we could use for calling the API, we ran a headless Firefox browser using Selenium. Even though we were able to actually obtain the token, we had issues related to performance, as running even headless instance of Firefox requires a lot of CPU power and RAM. This raised our costs and lowered our performance significantly enough to be a reason to purge this solution.

Even though without Selenium we were not able to obtain the token and therefore not able to call the API, we were able to obtain the HTML representing the website with listing's details by using only the httpx package and no tokens. The responses returning from the requests seemed to be more lengthy than we were expecting which prompted us to carefully investigate them. After a while, we figured that the HTML in the responses does not contain only the data showed on the web page, but also the raw API response in JSON.

In the end, we were able to extract the raw JSON response from the HTML, parse it and extract the respective listing's data from it.

3.1.6 Dependency Injection

Throughout the scraping bot's code, we wanted to use the DI design pattern for a multitude of reasons:

- helps to decouple the dependencies between components
- tends to result in a modular and reusable code
- improves the overall testability
- reduces the rippling effect of changes

In order to use DI throughout the whole code in a maintainable fashion, we decided to use a DI framework. We picked the `dependency_injector` python package, as it seemed to be popular based on multiple articles and also the number of stars its repository on GitHub got [54][55].

We used `dependency_injector` to create own class representing a DI container. Inside this container, we created all the singletons, factories etc. of classes that we wanted to use in the application and hid the injections and thus the dependencies inside it. This allowed us to maintain all the relationships between the classes and constructions of objects in one place, therefore making it easy to change the code without causing rippling effects throughout the application.

3.1.7 SQLite as cache

When we scrap new listings, check for price changes or mark listing presence/absence, we need to access already existing data that we have on the listings. These operations tend to happen quite frequently, quite often tens or hundreds of times per second. If we were to send a request, for example, each time we need to check if a listing with given id exists in our database, we would make possibly hundreds of requests per second. This could potentially slow down the API or even DOS it, as the used HW tends to be not very powerful as we are trying to keep the project as cost-efficient as possible.

That is why we created a SQLite DB which is an in-memory DB and therefore does not need a standalone server or even a running instance. This DB is used as a cache. When starting a new container with instance of scraping bot, the DB is created and loaded with IDs and prices of existing listings which are obtained from the API. After that, we maintain the cache throughout the run of the scraping bot and use it to check for existence of IDs and known listing prices. This results in a much smaller amount of requests main towards the API as we only need to communicate with it when finding a price change or adding/removing a listing.

3.2 Client bot

3.2.1 Sending images

When sending the notifications in form of embeds, we have the option to also send an image as part of the modal. This is very convenient as visuals play

a significant role when choosing a property. Therefore, we certainly do want to embrace this option and provide the users also with the image of property that is represented by the notification. When sending an image inside an embed, we pass in an image file in the form of stream of bytes. This data gets then stored at one of Discord's servers. This is optimal as we do not need to host the individual images and provide the embed with links to them.

At first, we wanted to use the image URL, that is part of the listing object, to fetch the image from a real estate portal. This approach worked fine until we needed to notify the user about a removal of listing. That is because at that point in time, the listing was no longer available at the real estate portal's website and therefore the link to the image we saved previously was no longer available either.

This led us to a different solution. We decided to download the images when scraping the listings' details and save them to an S3 bucket. The name under which we save the pictures is identical to the ID that the listing has in our database. Therefore, once a client bot receives a notification, it fetches the image not from the real estate portal's website, but rather from the S3 bucket. This allows us to provide the users with estate images even after the listing is removed.

One thing to consider is the cost of implementing the solution described above. For a model case, let's say that our application stores data on roughly 100k listings. That means we have to store 100k images in the S3 bucket. As the images we scrap tend to be smaller, their size is usually about 250KB. This totals out to about 25GB of storage needed to store said images. Using the AWS calculator [16] we can see that the price for storing such amount of data is roughly 0.5 dollars per month which is very little. This solution will therefore make little impact to our budget and is usable.

3.2.2 Modals vs Views

There are two ways that we can go about when collecting form data from users, modals and views. The original idea was to use modals as they feature a rather modern and simple look. In our opinion, working with modals is also more user friendly as they present themselves as a pop-up and once submitted disappear, thus not "spamming" the channel with unnecessary messages. Unfortunately, as modals are quite new to Discord, they at the moment support only simple text fields as the form of input. That means we are not able to add any buttons, suggestions, select menus or check-boxes. This is an issue as some of the information we need to collect from the user are for example the names of regions or counties that the subscription is supposed to be monitoring. Obtaining only valid values in those cases would be next to impossible and very error prone.

Due to the reasons above we decided to use views. Views are a more seasoned feature in Discord and allow buttons and select menus. They, however,

do not allow text inputs. That slightly complicates the implementation as we somehow need to obtain numerical information such as min/max prices and areas of estates that are supposed to be monitored. Such options have many possible values and it would be impossible to add them all to a select menu. Therefore, we created select menus that contain only some values that are sensible in given domain. For example, when filtering through houses, we expect that picking from area being smaller than 20 or 40 square meters will be granular enough for the vast majority of users. This is however something that will definitely be an important part for user testing.

With views there are also some drawbacks, the first one being only able to send five rows of elements in each view. This was solved by sending multiple views to the user as followups to the first one.

Another issue with views is, that they stay in channel even after they have been filled out and sent. We were able to solve this issue by saving the message which contains the view for each view and deleting these messages after successful completion of the form.

We also thought about combining modals and views. We would let users pick from the strictly defined options, such as region or county, using a select menu in view. For the other options with many possible values we would send the users a modal which they would fill out. In the end, we were not able to come up with a solution that would not feel confusing and unnatural, so we decided to stick with the views and predefined sets of values.

3.2.3 Regions and municipalities

When sending the users forms to fill out to create/update a subscription, we arrived at an issue related to the region and county select menus. In these menus, we need to offer the users values to pick from, however we do not really know which are the options that Nominatim can respond with. We would also like to have the regions and counties in exactly the same format as returned by Nominatim as it will allow us to match subscriptions to listings reliably and simply by string comparison. Therefore, we had to find a way to get all these options so we can offer them to the users.

At first, we tried to run a local instance of Nominatim server. In order to do so we used a containerized version available in this GitHub repository. After that, we connected inside the docker container using the Docker command line tool. Inside this container we were able to find the Postgres database which Nominatim's software uses to provide requested data. Unfortunately, the database was difficult to orient in and we were not able to obtain the data we were looking for.

Another option that we tried was downloading the OSM data of Czech republic from [here](#). After downloading said data, we used a Postgres database with extension called PostGIS which allows us to work with geographic data [56] and loaded the data into it using `osm2pgsql` tool. Using this database we were

able to fetch all the regions and all the counties, however we were not able to get the relationships between them.

At last, we tried a different option. We went to the website of state administration and pulled a list of all counties in Czech republic from there. We took the list and asked ChatGPT to parse it and provide us with GPS coordinates of one random point in each county. We took these points and passed them to a Python script which used Nominatim's reverse geocoding to get address for each of the points. From these addresses we were then able to pull all the counties, regions and their relationships and store them all in a database, thus allowing us to reuse them as needed.

3.3 REST API

3.3.1 Architecture

The API is split into multiple layers in order to properly separate the different concerns in the application. These layers are namely:

- Presentation layer
- Business layer
- Data-Access layer

As the name suggests, presentation layer is responsible for presenting information to users. Its another responsibility is to handle the user interactions and respond to them. In our API, the presentation layer is represented by controllers. The controllers consist of different endpoints which other applications use to communicate with the API. The controllers also convert the data in requests sent by other applications to the DTOs and call appropriate methods of classes from the business layer to accomplish the promised actions.

Business layer is where the logic of the application takes place. Some of its responsibilities tend to be data validations and calculation of all sorts. Business layer uses data-access layer to obtain and manipulate data in the database as it should not communicate with the database directly. In our case, the business layer consists of multiple different types of classes. These classes are services, assemblers, sender (classes responsible for sending message to AWS SQS queues). The classes which make use of rest of the classes in our business logic layer are services. They are the core classes that are communicated with by the controllers from presentation layer and also communicate with repositories from data-access layer.

Another layer of our API is the data-access layer. This layer is responsible for directly communicating with the database, requesting and saving data from/to it. It is communicated with by the classes from business layer. In our API, the data-access layer is in the form of repository interfaces.

3.3.2 Model conversion placement

One of the choices that had to be made is where to convert the entities to DTOs and vice-versa. Based on the architecture of our API, there were two options:

- Controllers
- Services

When researching which option is the preferred one, we were able to see that some sources prefer performing the conversion in controllers [57][58] while others prefer to do so in services [59][60]. Based solely on the amounts of opinions we went through, it seems like the topic of placing model conversions is quite controversial, yet placing it in controllers is usually preferred. At this point, we should also mention that the conversion logic is placed in completely separate classes. In this sections, placement of conversion simply discusses the place where methods of dedicated classes used for conversion are called.

Despite controllers being the more favored choice, we decided to perform the conversions in service for a multitude of reasons. Firstly, we feel like entities are parts of business layer and should not be shared outside of it to presentation layer which is exactly what would happen when performing the conversions in controllers. Even though it appears like this argument could also be made about DTOs not belonging to business layer, it might not be necessarily true. The DTOs always contain the data that we are comfortable sharing with the users. Therefore, passing such data to business layer does not cause any harm. On the other hand, entities might include sensitive information, such as database IDs and other information, which should not be shared with the users. Because of the stated reasons, sending entities with sensitive data out of the business layer of our application feels like "pulling the guts out" of our application.

Another reason as to why place the conversions into the service layer is to prevent cyclical dependencies. To elaborate, when placing the conversions into controllers, the services return, take in and work with entities only. This allows us to easily use one service inside another one. While this necessarily isn't a bad practice, it can lead to cyclical dependencies in the application which might require much refactoring to fix. This is why we decided to put the complex logic such as conversions, matching of listings to subscriptions or communication with AWS SQS into separate classes. Then we auto-wired said classes into the services and accomplished our goals using them. Therefore, our services themselves do not contain much detailed logic, but rather accomplish more complex tasks using methods/functionalities of said auto-wired classes. This allows us to perform changes to our application without causing much rippling effects. It also makes the whole process of writing tests much easier as different concerns are properly separated.

3.3.3 ModelMapper

Throughout the API we need to convert DTOs to entities and vice-versa. We can do this either manually or use a framework to help us. We decided to use a combination of these approaches. When converting DTOs to entities or the other way around, we use ModelMapper to map the simple attributes with identical names for us. By simple attributes we mean values that are of type Integer, String etc.

ModelMapper is a Java framework that is used for object mapping. We decided to use it mainly due to its good integration with Spring and also because it is very easy to use [61]. Model mapper saves us a many lines of code that is very prone to being affected by any changes made to the entities and DTOs. Even though once the application is release ready, changes in entities and DTOs do not happen very often, during the development, testing and beta phases of the project they can be quite frequent.

After that we also have to map some of the attributes manually as they are more complex. When mapping DTOs to entities, the DTOs tend to contain the IDs of objects that are nested in the entities. Therefore, we are using repositories of the nested objects' classes to get the instances of said objects that feature the looked for IDs. We then set these instances as attributes to the entity we are converting DTO to.

When mapping entities to DTOs, the manual mapping is simpler as it does not require us to use repositories. In order to set the attributes of DTOs that contain IDs, we simply use ID getters of the corresponding entities' attributes.

3.3.4 Hibernate Validator

When creating or updating entities, we need to check that the entities match certain constraints, such as attributes being not null. In order to do so, we decided to use Hibernate Validator. Hibernate Validator is library that allows us to validate entities in a standardized manner using annotation-based constraints. It therefore helps us ensure data integrity and consistency. Using Hibernate Validator rids us of the responsibility of creating validation classes and manually checking whether entities match all constraints [62].

3.3.5 Access control

The API is supposed to be used solely by other parts of our application, namely the scraping bot and the client bot. Therefore, it should not be directly accessible by the public at all. At first we planned to use Spring Security and some form of authentication such as credentials or certificates. This would add more complexity to the API and would also require us to manage said secrets.

That is why we came up with a simpler solution. As we are using AWS to host our application we can create a VPC in which all our resources will be

running. We can also assign security groups to resources, that dictate which traffic is allowed to access the given resource and which is not. That is exactly what we did with the resources, that are running our API. We assigned them all a security group that is set to not allow any inbound traffic that did not originate in our VPC. This way, we do not need to manage any credentials or certificates and can also be certain, that no one outside of our VPC can access the API.

3.3.6 Price change

At first we planned to create a POST endpoint for creating PriceChange entities and saving them into the database. While implementing the endpoint, we realised that if when price change is registered, it also means that the price of the related RealEstate entity should be changed. As RealEstate entities can be manipulated using appropriate endpoints, doing so as a side-effect when creating a different object seemed inappropriate to us. That is why it was decided to purge the endpoint used for creating PriceChange entities and create them when the price is updated using the RealEstate entity's PUT endpoint. This seemed to be a better solution as the PriceChange is not an entity deemed for users, but rather for our analytical purposes.

3.3.7 Deriving missing parts of addresses from existing data

When scraping the real estate listings, we can see that the addresses miss the county or even the municipality attributes at times. We tried to analyse the addresses that miss said attributes and were able to see a pattern which these addresses tend to follow.

The most important pattern we were able to spot, is that all the addresses located in Prague always miss both the municipality and the county. The state of these addresses, however, was always *Prague* and therefore the solution of missing county was simply also setting it to *Prague*. It makes sense that for Prague the municipality may be missing, however it would be nice to always have there the respective district, such as *Prague 1*, *Prague 2* etc. Thankfully, we were able to see that the suburb of said addresses is always set and it is always possible to tell which district it belongs to. Based on these information, we pulled a list of districts with their respective suburbs from the web page of the Prague capital city. Then we proceeded to parse this raw data using OpenAI's ChatGPT and created a database table containing mapping of suburbs to districts. Afterwards, we were able to use this table to set the correct district as the addresses municipality in our code.

There were two more scenarios in which the county was missing. In these scenarios, the attribute *state* was always set to either *střední čechy* or *jihozápad*. For *střední čechy* we set the county value to *Středočeský kraj*. After analysing the data further, we were able to figure out that when the *state*

attribute is set to *jihozápad* and the *county* attribute is blank, the missing value is always *Jihočeský kraj*.

Thanks to completing the missing values the way described above, we are now able to match more real estate listings to subscriptions and can also provide the users with greater granularity in Prague region when creating subscriptions.

3.4 AWS

3.4.1 Multiple SQS queues VS. Message Groups

There are two ways as to how we can use AWS SQS when sending/consuming messages:

- Multiple queues, one message group per queue
- Single queue, multiple message groups per queue

Having multiple queues does not incur additional costs, as AWS SQS is a pay-as-you go model. Therefore, we pay only for the amount of operations we perform towards AWS SQS, but do not pay a fee per existing queue. The drawback of this approach is, that we need to store and update multiple queue ARNs.

The other option is to have only a single queue, that will feature multiple message groups. This means that we only have to take care of one queue and thus only of one ARN. It also means, that we do not have to check multiple queues, but only one. This will save us a lot of operations and therefore a lot of resources. This will allow us to check for new messages/notifications much more frequently at the same price. The only drawback is, that we need to store a list of message group IDs between two applications. One option would be to create a table or enumerable in database and fetch this information from there. This would be a great solution as the message groups could be updated just from one place and change would propagate to all the applications. However, as there are only two applications, REST API and Client bot, that work with this queue and only three message groups, we decided to create enumerable in each of the applications and keep them in check through versioning. This approach will also save us some resources as we will make less requests to database/API.

3.4.2 ECS Task definition

When setting up an ECS Task definition, there are multiple things that we need to carefully configure in order for our application to behave the way we want it to.

3. IMPLEMENTATION

First of all, it is very important to set up the network mode to Bridge instead of VPC. In our case, the inbound/outbound setting of the VPC are very strict, as we do not want the API endpoints to be accessible outside of VPC. By setting the network mode to bridge, we allow the containerized instance to use the network of the EC2 instance that it is running on.

Second of all, we need to pass the scraping and client bot a Discord bot token, that will allow them to act as the respective bots. As this is a secret value, we do not want it featured inside the code or the repository. ECS Task definition allows us to solve this issue by passing it an ARN of the secret stored in AWS Secret Manager that hides the value of the tokens. However, the ARN has to be passed during the creation of ECS Task definition as there is no option to later pass it to individual tasks.

3.4.3 ECS Service

When setting up an ECS service, we have to carefully adjust some parameters in order to allow deployments of new docker containers with the little resources that we are running. These parameters are called desired tasks, minimum and maximum running task thresholds. Desired tasks tells the service how many instances we ideally want to be running. Minimum/maximum running tasks threshold, tells the service, how many percent of the desired tasks may be running at minimum/maximum. When leaving the thresholds at default values, 100% and 200% respectively, we were not able to successfully deploy new versions of our applications. After reading through the logs we were able to see, that the issue has to do with too little resources to run another task. We were able to solve this issue by setting the thresholds at 0% and 100% respectively, allowing the service to stop the existing task and after that start the new one, thus temporarily running zero tasks even though the desired tasks were set to 1.

3.4.4 Deployment

In order to deploy our application to AWS we decided to use GitHub Actions. GitHub already features multiple templates for the most common Actions and we were lucky enough to find one for deploying an application to ECS, which required only minor changes. In order to use the Action, we also had to provide it with AWS Credentials as well as ARNs of resources such as ECS Task Definition, ECS Service or ECR Repository. All of these information are sensitive and we do not want to make them publicly visible. That is why we used GitHub Secrets to store all this information and provided them to the Action through references to said Secrets.

The Action performs multiple steps when running. First, it builds Docker image of the application. Next, it pushes the created Docker image into its

respective repository in ECR. After that, it instructs the ECS Service to pull the new image from ECR and replace the currently running tasks with it.

One more thing that we changed in the Action template are the triggers. By default, the Action would get triggered by every push to the main branch. As we prefer to have more control over the deployment of new versions, we updated the Action so it can be triggered only manually by us. This way, we can double check that everything got merged the way it was supposed to, when pushing code to the main branch and after that deploy the new version manually.

Testing

4.1 Types of used tests

4.1.1 Unit tests

The point of unit tests is to test the smallest piece of code that can be logically isolated. This tends to be method, function or a property in the majority of programming languages. Unit tests are written during the development and they are capable of detecting bugs in software early on. This is important as fixing the bugs right away is simpler than finding and fixing them down the road when the software is bigger [63][64].

4.1.2 User tests

User testing is a process during which real users test the interface and functionalities of given application in real world conditions. User testing can not only show us whether or not the application is working, but also how comfortable, intuitive and easy to use it is to the users. These are very important metrics as they can decide the fate of our application, meaning whether or not will people use it [65].

4.2 Client bot

4.2.1 User testing

The main method used to verify that the client bot is working as expected and that the UX is great were user tests. The users which tested the application were picked from our colleagues and friends. The users were instructed to perform the following tasks.

4.2.1.1 Create subscription

The users were tasked to create subscription that is set to monitor real estates that:

- are houses
- are in Okres Nymburk
- are in the price range between one and three millions
- are bigger than 50 sqm. and smaller than 100 sqm.
- feature a pool

This task allowed us to collect feedback on the way filters are displayed and about the options in select menus regarding prices and areas. For the same reason we also purposefully tasked the users with selecting a value that was not in the select menu options to see how they reacted to it and later questioned them about it.

All the users were able to determine that the *sub* command is used to create a new subscription. Some of the users, the ones with at least basic knowledge of English, were able to determine the functionality of the command based on its name. The rest was able to figure it out from the description of the command. The way users were able to determine the functionality of the command was the same for all the tested commands and therefore will not be mentioned further.

Even though some of the users we slightly confused by not having the exact option in the menu, they were all able to solve the situation by selecting the closest less strict option. When asked about the granularity of options, all users were satisfied with the options regarding the area sizes. On the other hand, some users were not satisfied with the options in price select menus and suggested that prices upwards to ten million should more granular. In reaction to said critique, we added more options to the select menu. As a result there are now also values 6, 7, 8 and 9 millions to pick from.

Some of the users were also not satisfied with the way the resulting subscription that they created was presented to us, especially calling out its poor design and readability. This was an understandable critique which was admittedly slightly expected when starting the user testing. In response to said critique, we created and embed which conveys the information about the created subscription in a more readable and graphically more pleasing manner.

One thing that we were slightly afraid of was the users critiquing the way the form for creating a subscription is presented to them. This would a close to impossible challenge to solve for us, due to Discord API not yet being capable of anything more than what we are currently doing in terms of sending views. Thankfully none of the users were dissatisfied with the way subscription create form is presented to them and therefore there is no reason to change it.

4.2.2 Update subscription

The users were tasked to update the subscription created in 4.2.1.1. They were instructed to change the subscription to monitor real estate that:

- are apartments
- are in Okres Karlovy Vary
- are in the price range between two and five millions
- are bigger than 70 sqm. and smaller than 180 sqm.
- feature a garage

This task ensured that all the previously set values can be changed and that the process of deselecting the previous values is comfortable to the users.

All the users were able to determine, that the command *sub_update* is used to update existing subscriptions. Everyone was also capable to select the requested subscription from the returned drop-down menu without any issues.

As for changing the already set values, none of the users had any issues doing so. Some of the users also noted that it is great that when updating the subscriptions all filters get shown by default. When asked why, the users said that they would otherwise expand all the filters anyway as they would anyway want to check that the pre-selected values are set correctly.

4.2.3 View subscription details

The users were tasked to view the details of the subscription they just created.

All the users were able to determine, that the command *sub_detail* is the one to use to view the subscriptions details. As the interface of this command is identical with the first part of update subscription command, we did not ask the users about the experience with the UI and included this command just to check it functions as intended.

4.2.4 Remove subscription

The users were tasked to remove the subscription that they created and edited in the previous tasks.

All the users were able to determine that the command *sub_delete* is used for removing the subscription. As the drop-down menu with existing subscriptions was the same one as the one returned during the update, the users were not questioned on it again.

4.2.5 Verify the bot is working

The users were tasked to check whether the bot is online and responding to their commands without altering any subscriptions. They were not provided any more instructions as to how to accomplish this goal as we wanted to see whether the command

ping is simple and intuitive enough.

All the users that tested the application were common with the meaning of ping and therefore were right away confident in what the ping command is for and were not afraid to use it right away.

4.3 REST API

The whole API code was covered using unit tests. In order to implement these tests, we used multiple different frameworks.

- JUnit
- JaCoCo
- Mockito

JUnit is a Java framework which allows to implement unit tests in a standardized manner using annotations. It provides us with many useful features such as initializing the variables and mocks before each test or cleaning up the environment after every test and much more.

JaCoCo is a library that does not help us with writing tests per se, but rather with analyzing existing tests and seeing how well they cover the code. These data are conveyed to us using a simple UI in the form of website which is also generated by JaCoCo. Using JaCoCo can help us spot undertested places in our code and make sure that all the possible cases are properly covered by the unit tests.

Mockito is a library that allows us to create mocks of different classes. This is very important, especially in unit testing, as the tests are supposed to focus on just the tested method and nothing else. By injecting the tested classes and therefore their methods with mocks instead of real instances, we can be sure that we are really testing only the code we are supposed to. It also allows to monitor whether the tested method called the mocks expected amount of times, or if the values that the method returns correspond with the values provided by the mocks.

Conclusion

The goal of this thesis was to implement an application which is capable of performing multiple tasks. The application should be able to scrap listings of real estates for sale on Czech real estate market. It should also be able to communicate with its users through the Discord application and provide them with the means to create subscriptions. These subscriptions should allow the users to monitor the Czech real estate market for listings that match the properties of their subscriptions. Another goal of this thesis was to ensure the applications availability and scalability by deploying it to cloud. All of the previously stated goals were successfully reached and the resulting application fulfills all of the functional and non-functional requirements defined in chapter 1.

In the chapter number one 1, the existing solutions for the solved problem were analysed. It was concluded that although some similar solutions exist, they tend to be pricey, cover less real estate portals, are no longer maintained and/or not support Discord. The functional and non-functional requirements as well as the use cases were also defined in this chapter.

In the second chapter 2, it was decided that the application is going to be split into three parts. These parts are namely scrapers, client bot and API. For each of these parts we discussed which technologies are the most appropriate ones to use. It was also stated which cloud provider is going to be used to host the application. In the end, we designed the high-level architecture of the application which displays what cloud services are used as well as the use of other external services. It also discusses the relationships and communication between different parts of the application.

The third chapter 3 discusses the implementation of the whole application, which was based on the analysis and design from chapters 1 and 2. The problems that we came across and the solutions and fixes used to solve them are described there. Also, the used cloud services and deployment of the application are discussed to a greater detail in this chapter.

Chapter number four 4 is devoted to testing. Different types of used tests

are explained in this chapter. Also the details of testing of each of the parts of the application are discussed here, especially the details of user testing of the client bot as it revealed some imperfections which were then addressed.

In conclusion, we would like to say that we deem working on this thesis highly beneficial to us. Throughout the thesis, we got to deepen our AWS skills, especially when working with ASG and ECS. We also learnt to use the Discord API, work with Nominatim and with the Python's asyncio package. The knowledge we gained was also not limited to the technical domain, as we got to understand and orient ourselves in the real estate market which we also view as an invaluable skill for our life.

5.1 Visions for the future

As the topic of this thesis was chosen due to our personal interest, we plan to continue the development of this application in the future. At the moment, the application is in Beta version. Once we are confident with its state we plan to make it available through a website of our own. We also understand that not all potential users are also users of Discord and we recognize the opportunities that providing our notifications through other platforms than Discord offers us. That is why the whole application was implemented with extendability in mind so that email or phone notifications can be easily added in the future, thus broadening our target group.

Bibliography

1. NUMBEO. *Europe: Current Property Prices Index by City* [online]. [visited on 2023-06-28]. Available from: https://www.numbeo.com/property-investment/region_rankings_current.jsp?region=150.
2. REALITNÍ HLÍDACÍ PES. *Realitní hlídací pes* [online]. [visited on 2023-02-19]. Available from: <https://www.realitnihlidacipes.cz/>.
3. REALITYMON. *realitymon* [online]. [visited on 2023-02-19]. Available from: <http://www.realitymon.cz/#>.
4. INFOEXE S.R.O. *infoexce* [online]. [visited on 2023-02-19]. Available from: <http://infoexe.cz/infoexce>.
5. GURU99. *What is a Functional Requirement in Software Engineering?* [Online]. 2023-01. [visited on 2023-02-24]. Available from: <https://www.guru99.com/functional-requirement-specification-example.html>.
6. NUCLINO. *A Guide to Functional Requirements (with Examples)* [online]. [visited on 2023-02-24]. Available from: <https://www.nuclino.com/articles/functional-requirements>.
7. SAGGU, Ashmeet. *Non-functional Requirements in Software Engineering* [online]. 2023-02. [visited on 2023-02-24]. Available from: <https://www.geeksforgeeks.org/non-functional-requirements-in-software-engineering/>.
8. ALTEXSOFT. *Non-functional Requirements: Examples, Types, How to Approach* [online]. 2022-07. [visited on 2023-02-24]. Available from: <https://www.altexsoft.com/blog/non-functional-requirements/>.
9. POSEY, Brien. *Top public cloud providers of 2023: A brief comparison* [online]. 2022-11. [visited on 2023-02-25]. Available from: <https://www.techtarget.com/searchcloudcomputing/tip/Top-public-cloud-providers-A-brief-comparison>.

BIBLIOGRAPHY

10. INFLECTRA. *Use Cases & Scenarios: What They Are & More* [online]. 2023-01. [visited on 2023-02-25]. Available from: <https://www.inflectra.com/Ideas/Topic/Use-Cases.aspx>.
11. DALY, Nicky. *What Is a Use Case?* [Online]. 2022-05. [visited on 2023-02-25]. Available from: <https://www.wrike.com/blog/what-is-a-use-case/>.
12. BRUSH, Kate. *use case* [online]. [visited on 2023-02-25]. Available from: <https://www.techtarget.com/searchsoftwarequality/definition/use-case>.
13. AWS. *Amazon S3 pricing* [online]. [visited on 2023-05-10]. Available from: <https://aws.amazon.com/s3/pricing/>.
14. CHAND, Mahesh. *Most Popular Databases In The World (2023)* [online]. 2022-12. [visited on 2023-03-18]. Available from: <https://www.c-sharpcorner.com/article/what-is-the-most-popular-database-in-the-world/>.
15. AWS AMAZON. *What is NoSQL?* [Online]. [visited on 2023-03-18]. Available from: <https://aws.amazon.com/nosql/>.
16. AWS. *Amazon AWS pricing calculator* [online]. [visited on 2023-05-10]. Available from: <https://calculator.aws/#/>.
17. SCALEGRID. *PostgreSQL vs. Oracle: Difference in Costs, Ease of Use & Functionality* [online]. 2020-07. [visited on 2023-03-18]. Available from: <https://scalegrid.io/blog/postgresql-vs-oracle-difference-in-costs-ease-of-use-functionality/>.
18. AWS AMAZON. *Amazon RDS for PostgreSQL Pricing* [online]. [visited on 2023-03-18]. Available from: <https://aws.amazon.com/rds/postgresql/pricing/>.
19. AWS AMAZON. *Amazon RDS for SQL Server Pricing* [online]. [visited on 2023-03-18]. Available from: <https://aws.amazon.com/rds/sqlserver/pricing/>.
20. TIOBE. *TIOBE Index for March 2023* [online]. 2023-01. [visited on 2023-03-12]. Available from: <https://www.tiobe.com/tiobe-index/>.
21. MITCHELL, Brad. *The 5 Easiest Programming Language to Learn (and Why)* [online]. 2022-11. [visited on 2023-03-12]. Available from: <https://www.codingdojo.com/blog/easiest-programming-language-to-learn>.
22. PROXIES, Scraping. *The Best Language for Web Scraping (Hint: There are 5)* [online]. 2021-12. [visited on 2023-03-12]. Available from: <https://lightproxies.com/blog/best-language-for-web-scraping/>.

23. DILMEGANI, Cem. *Best Web Scraping Programming Languages in 2023 with Stats* [online]. 2023-03. [visited on 2023-03-12]. Available from: <https://research.aimultiple.com/web-scraping-programming-languages/>.
24. PROMPT CLOUD. *What are The Best Programming Languages for Web Scraping?* [Online]. 2017-08. [visited on 2023-03-12]. Available from: <https://www.promptcloud.com/blog/best-programming-language-for-web-scraping/>.
25. JAMES, Alexander. *Best Programming Languages for Web Scraping* [online]. 2023-02. [visited on 2023-03-12]. Available from: <https://scrape.do/blog/best-programming-languages-for-web-scraping/>.
26. ZENROWS. *Top 5 JavaScript and NodeJS web scraping libraries in 2023* [online]. 2022-09. [visited on 2023-03-12]. Available from: <https://www.zenrows.com/blog/javascript-nodejs-web-scraping-libraries#playwright>.
27. GEEKSFORGEEKS. *What is Callback Hell and how to avoid it in Node.js ?* [Online]. 2022-03. [visited on 2023-03-12]. Available from: <https://www.geeksforgeeks.org/what-is-callback-hell-and-how-to-avoid-it-in-node-js/?ref=rp>.
28. LAI, Hongli. *What causes Ruby memory bloat?* [Online]. 2019-03. [visited on 2023-03-12]. Available from: <https://www.joyfulbikeshedding.com/blog/2019-03-14-what-causes-ruby-memory-bloat.html>.
29. MATUSZ, Jan. *Ruby Quirks* [online]. 2019-09. [visited on 2023-03-12]. Available from: <https://www.visuality.pl/posts/ruby-quirks>.
30. RICHTER, Felix. *Amazon, Microsoft & Google Dominate Cloud Market* [online]. 2022-12. [visited on 2023-03-05]. Available from: <https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/>.
31. SERVERLESS FRAMEWORK. *Serverless Infrastructure Providers* [online]. [visited on 2023-03-12]. Available from: <https://www.serverless.com/framework/docs/providers>.
32. SERVERLESS FRAMEWORK. *AWS Python Example* [online]. [visited on 2023-03-12]. Available from: <https://www.serverless.com/examples/aws-python>.
33. BESWICK, James. *Using Amazon EFS for AWS Lambda in your serverless applications* [online]. 2020-06. [visited on 2023-03-11]. Available from: <https://aws.amazon.com/blogs/compute/using-amazon-efs-for-aws-lambda-in-your-serverless-applications/>.
34. MICROSOFT. *Storage considerations for Azure Functions* [online]. 2023-02. [visited on 2023-03-11]. Available from: <https://learn.microsoft.com/en-us/azure/azure-functions/storage-considerations?tabs=azure-cli>.

35. AMAZON AWS. *How do I give internet access to a Lambda function that's connected to an Amazon VPC?* [Online]. 2023-03. [visited on 2023-03-11]. Available from: <https://aws.amazon.com/premiumsupport/knowledge-center/internet-access-lambda-function/>.
36. MICROSOFT. *Azure Functions networking options* [online]. 2018-10. [visited on 2023-03-11]. Available from: <https://learn.microsoft.com/en-us/azure/azure-functions/functions-networking-options?tabs=azure-cli#virtual-network-integration>.
37. ASHAN. *Can an Azure Function access internet?* [Online]. 2018-10. [visited on 2023-03-11]. Available from: <https://stackoverflow.com/questions/53394481/can-an-azure-function-access-internet>.
38. AMAZON AWS. *How do I troubleshoot Lambda function invocation timeout errors?* [Online]. 2023-02. [visited on 2023-03-11]. Available from: <https://aws.amazon.com/premiumsupport/knowledge-center/lambda-troubleshoot-invocation-timeouts/>.
39. AMAZON AWS. *Azure Functions hosting options* [online]. 2022-11. [visited on 2023-03-11]. Available from: <https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale>.
40. MICROSOFT. *Azure Functions pricing* [online]. [visited on 2023-03-11]. Available from: <https://azure.microsoft.com/en-us/pricing/details/functions/>.
41. MICROSOFT. *Windows Virtual Machines Pricing* [online]. [visited on 2023-03-11]. Available from: <https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/>.
42. WIKIPEDIA. *cron* [online]. 2023-03. [visited on 2023-03-11]. Available from: <https://en.wikipedia.org/wiki/Cron>.
43. RAMANUJAM, Sriram. *How to Run a Cron Job Inside a Docker Container?* [Online]. 2022-08. [visited on 2023-03-11]. Available from: <https://www.baeldung.com/ops/docker-cron-job>.
44. DISCORD.PY. *discord.ext.tasks - asyncio.Task helpers* [online]. [visited on 2023-03-11]. Available from: <https://discordpy.readthedocs.io/en/stable/ext/tasks/index.html>.
45. EMERSON, Rob. *What Language Are Discord Bots Written In?* [Online]. 2023-01. [visited on 2023-03-12]. Available from: <https://www.itgeared.com/what-language-are-discord-bots-written-in/>.
46. WRITEBOTS. *How to Make a Discord Bot in 2023* [online]. 2023-01. [visited on 2023-03-12]. Available from: <https://www.writebots.com/how-to-make-a-discord-bot/>.

47. GIRDHAR, Abhinav. *Appy Pie : What language are discord bots written in?* [Online]. 2022-04. [visited on 2023-03-12]. Available from: <https://www.appypie.com/faqs/what-language-are-discord-bots-written-in>.
48. STATISTICS&DATA. *Most Popular Backend Frameworks – 2012/2023* [online]. [visited on 2023-03-19]. Available from: <https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2023/>.
49. KAUR, Preet. *10 Most Popular Frameworks For Building RESTful APIs* [online]. 2022-12. [visited on 2023-03-19]. Available from: <https://www.moesif.com/blog/api-product-management/api-analytics/10-Most-Popular-Frameworks-For-Building-RESTful-APIs/>.
50. PYCODEMATES. *The Top Ten Web Frameworks for creating REST APIs -Backend Development* [online]. 2022-08. [visited on 2023-03-19]. Available from: <https://www.pycodemates.com/2022/08/top-ten-web-frameworks-for-creating-rest-apis.html>.
51. ZHANG, Mary. *Top 10 Cloud Service Providers Globally in 2023* [online]. 2023-01. [visited on 2023-03-05]. Available from: <https://dgtlinfra.com/top-10-cloud-service-providers-2022/>.
52. OSMF. *Nominatim Usage Policy (aka Geocoding Policy)* [online]. [visited on 2023-05-07]. Available from: <https://operations.osmfoundation.org/policies/nominatim/>.
53. SEZNAM.CZ, A.S. *Sreality* [online]. [visited on 2023-02-19]. Available from: <https://www.sreality.cz/>.
54. KALKMAN, Patrick. *Dependency Injection in Python* [online]. 2023-04. [visited on 2023-06-17]. Available from: <https://itnext.io/dependency-injection-in-python-a1e56ab8bdd0>.
55. ETS-LABS. *python-dependency-injector* [online]. [visited on 2023-06-17]. Available from: <https://github.com/ets-labs/python-dependency-injector>.
56. POSTGIS PSC & OSGEO. *About PostGIS* [online]. [visited on 2023-06-22]. Available from: <http://postgis.net/>.
57. BAELDUNG. *Entity To DTO Conversion for a Spring REST API* [online]. 2022-12. [visited on 2023-06-27]. Available from: <https://www.baeldung.com/entity-to-and-from-dto-for-a-java-spring-application>.
58. KHAN, Nadeem. *Best Practices in Spring Boot Project Structure* [online]. 2021-06. [visited on 2023-06-27]. Available from: <https://medium.com/learnwithnk/best-practices-in-spring-boot-project-structure-layers-of-microservice-versioning-in-api-cadf62bd3459>.

BIBLIOGRAPHY

59. FADATARE, Ramesh. *Spring Boot DTO Example Tutorial* [online]. [visited on 2023-06-27]. Available from: <https://www.javaguides.net/2022/12/spring-boot-dto-example-tutorial.html>.
60. J.G., Daniel. *Using DTO to transfer data between service layer and UI layer* [online]. [visited on 2023-06-27]. Available from: <https://stackoverflow.com/questions/16866102/using-dto-to-transfer-data-between-service-layer-and-ui-layer/16872129#16872129>.
61. MODELMAPPER. *modelmapper* [online]. [visited on 2023-06-28]. Available from: <https://modelmapper.org/>.
62. HIBERNATE. *Hibernate Validator* [online]. [visited on 2023-06-28]. Available from: <https://hibernate.org/validator/>.
63. TECHTARGET. *unit testing* [online]. [visited on 2023-06-28]. Available from: <https://www.techtarget.com/searchsoftwarequality/definition/unit-testing>.
64. SMARTBEAR. *What Is Unit Testing?* [Online]. [visited on 2023-06-28]. Available from: <https://smartbear.com/learn/automated-testing/what-is-unit-testing/>.
65. OMNICONVERT. *What is... User testing* [online]. [visited on 2023-06-28]. Available from: <https://www.omniconvert.com/what-is/user-testing/>.

Acronyms

SLA	Service level agreement
GUI	Graphical user interface
REST	Representational state transfer
GCP	Google Cloud Platform
AWS	Amazon Web Services
API	Application Programming Interface
CPU	Computer Processing Unit
RAM	Random Access Memory
VPC	Virtual Private Cloud
NAT	Network Address Translation
EC2	Elastic Compute Cloud
EFS	Elastic File System
UI	User Interface
UX	User Experience
RDBMS	Relational Database Management System
DI	Dependency Injection
EBS	Elastic Beanstalk
ECS	Elastic Container Service
ECR	Elastic Container Registry

A. ACRONYMS

ARN Amazon Resource Name

DTO Data Transfer Object

ID Identifier

DB Database