Mobile application for booking tickets

Bc. Jozef Bugoš
Study program: Open Informatics
Branch: Software Engineering

May 2022
Supervisor: Ing. Božena Mannová, Ph.D.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Bugoš Jozef**	Personal ID number:	**466219**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

Mobile application for booking tickets

Master’s thesis title in Czech:

Mobilní aplikace pro rezervaci jízdenek

Guidelines:

The topic of this thesis is to solve the situation when it is not possible to buy a ticket, because they are all sold-out. In this situation, it is necessary to check whether somebody did not cancel a ticket or the provider has not increased the capacity. The outcome of this thesis should be a mobile application that automates the necessary activities.
Analyze network communication of selected reservation systems. Design and implement a server to emulate their communication. Create a back-end service that will emulate this communication and provide the necessary data for the proposed mobile application. Design a cross-platform mobile application that will use this service. Design the architecture of the entire project and find suitable technologies for implementation. Justify your decisions. Implement the application. Perform system functionality testing and evaluate the test results. Suggest possible improvements and extensions to the application. Use software engineering resources for processing.

Bibliography / sources:

1. Roger S. Pressmann Bruce Maxim: Software Engineering: A Practitioner's Approach , ISBN-10: 9780078022128
2. Spring Framework. https://spring.io/projects/spring-framework.
3. What Is a REST API? https://www.akana.com/blog/what-is-rest-api.

Name and workplace of master’s thesis supervisor:

Ing. Božena Mannová, Ph.D. Center for Software Training FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **14.02.2022** Deadline for master's thesis submission: **20.05.2022**

Assignment valid until: **19.02.2024**

Ing. Božena Mannová, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature

Acknowledgement / Declaration

I would like to thank Ing. Božena Mannová, Ph.D., the supervisor of this master thesis for her valuable advice and guidance, which was very helpful.

I declare, that I have done the assigned project alone led by the supervisor. I used only literature, which is listed in this work. In Prague 10. 5. 2022

. . . .

Abstrakt / Abstract

Tento projekt sa zaoberá analýzou problému kúpy lístkov v prípade, že sú vypredané, podrobnejšie pre poskytovateľov RegioJet a FlixBus. Navrhuje riešenie pozostávajúce z mobilnej aplikácie fungujúcej ako používateľské rozhranie a zdroj notifikácii a serveru schopného emulovať API vybraných poskytovateľov a pravidelne sledovať zmeny v dostupnosti lístkov. Výsledkom tohto projektu je implementácia vyššie uvedeného riešenia s funkčným serverom a mobilnou aplikáciou nasadenou na Google Play Store.

Kľúčové slová: Verejná doprava, Systém sledovania dostupnosti lístkov, RegioJet, FlixBus, Spring, Java, Flutter, Dart, Mobilná aplikácia, Vývoj naprieč platformami

This project deals with analyzing the problem of trying to buy tickets when they are sold out, in more detail for providers RegioJet and FlixBus. It proposes a solution consisting of a mobile application for user interface and notifications, and a backend able to emulate API of chosen providers and periodically watch for changes in the availability of the tickets. The result of this project is the implementation of the aforementioned solution with a working server and mobile application functioning and deployed on Google Play Store.

Keywords: Transit, Ticket Availability Tracking System, RegioJet, FlixBus, Spring, Java, Flutter, Dart, Mobile Application, Cross-Platform Development

Contents /

1 Introduction 1
2 Analysis 2
2.1 Target audience 2
2.2 Network communication 2
2.3 Current system 5
2.4 Goals 6
2.5 Components 6
2.6 Conclusion 6
3 Server-side 7
3.1 Technology 7
3.2 Database 7
3.3 Architecture 8
3.4 Conclusion 8
4 Implementation 9
4.1 Design 9
4.2 Code 10
4.2.1 Common code base 10
4.2.2 Provider's specifics 14
4.2.3 Caching 15
4.2.4 RESTful API 17
4.2.5 Database 17
4.2.6 Watchers 20
4.3 Conclusion 20
5 Watchers 21
5.1 Overview 21
5.2 Scheduled 21
5.3 Structure 22
5.4 Examples 24
5.5 Conclusion 28
6 Future work and deployment 29
6.1 Deployment 29
6.2 Future features 31
6.3 Conclusion 32
7 Introduction to the mobile application 33
8 Technology 34
8.1 Cross-platform development 34
8.2 Frameworks 34
8.3 Conclusion 35
9 Application design 36
9.1 Pages 36
9.1.1 Splash screen 37
9.1.2 Details and routes lists 38
9.1.3 Search 40
9.1.4 Seat classes 41
9.1.5 Watchers 42
9.1.6 Settings 43
9.1.7 Watched routes 44
9.2 Desing vs RegioJet's web 44
9.3 Localization 47
9.4 Conclusion 48
10 User Testing 49
10.1 Form 49
10.1.1 Participant 1 49
10.1.2 Participant 2 50
10.1.3 Participant 3 50
10.1.4 Participant 4 50
10.2 Summary 50
10.3 Conclusion 51
11 Future work and deployment 52
11.1 Publishing 52
11.1.1 Google Play Store 52
11.1.2 App Store 52
11.2 Future features 53
11.3 Promotion 54
11.4 Conclusion 54
12 Conclusion 55
A Symbols 57
References 58

/ Figures

2.1. RegioJet request for locations . . . 3
2.2. RegioJet response with locations . . . 3
2.3. RegioJet request for routes . . . 3
2.4. RegioJet response with routes . . . 4
2.5. RegioJet request for route's details . . . 4
2.6. RegioJet response with route details. (Red are free seats overall, green are free seats in the given class) . . . 5
4.1. Layered API Architecture . . . 9
6.1. Deployment docker . . . 30
8.1. Trend on stackoverflow . . . 35
9.1. Splash screen page . . . 37
9.2. Parameter details without routes . . . 38
9.3. Parameter details with routes . 39
9.4. Search Page . . . 40
9.5. Seat classes page . . . 41
9.6. Watchers page . . . 42
9.7. Settings page . . . 43
9.8. Watched routes page . . . 44
9.9. App's locations search . . . 45
9.10. RegioJet's locations search . . . 45
9.11. App's tariffs . . . 45
9.12. RegioJet's tariffs . . . 45
9.13. App's connection details . . . 46
9.14. RegioJet's connection details . . 46
9.15. Routes in English . . . 47
9.16. Settings in English . . . 48
11.1. App running on Mac . . . 53

Chapter 1
Introduction

Public transportation is a sector that has experienced massive bloom in the years before the pandemic. After many years of stagnation, arrival of the non-state transport providers (hereinafter referred to as private providers), such as RegioJet[1], LeoExpress[2] or FlixBus[3], started competition not only between themselves, but forced the state transport providers (hereinafter referred to as state providers) to rose up from their sleep.

Private providers quickly overcame their state counterparts in popularity, as they were considered a „fresh breath“ in public transportation. Until then, most of the vehicles used by the state providers were old and not in a good state, being used continuously for many years. Due to this, the common view of public transport was that of a necessary evil, one you had to endure from time to time out of necessity.

This changed when private providers came with their own machinery, not necessarily newer in age, but remodeled and better equipped. As the price was still comparable to one of the state providers, it was only logical that people started using mostly their services. This trend continued, even though the state providers tried to adapt to the new standard of transportation until it reached a point where the tickets for holidays or even weekends were sold out weeks in advance.

With this came the need to buy tickets as soon as they were available. However, what was one to do if the tickets were already sold out? One could only periodically visit a web page of the provider and check whether someone canceled their ticket or the provider increased the capacity.

In our thesis, our primary goal is to address the aforementioned issue. We will show that automating this process in form of a mobile application brings great value. Firstly, we will have to analyze the network communication of chosen reservations systems. Secondly, to create a back-end service that would emulate this communication, expose their API through an adapter for our mobile application and periodically check the watched connections for changes. Lastly, to create a cross-platform mobile application that would use this service and enable users to select what connections to watch via a familiar and intuitive user interface.

This project will consist of two main parts. The first one, contained in the first half of the thesis, will be dealing with the analysis of network communication and the creation and deployment of the back-end service. The second one will be describing the development of the mobile application, its design, future work, and deployment in the applications stores, namely Google's Play Store and Apple's App Store.

[1] `https://www.regiojet.com/`
[2] `https://www.leoexpress.com/en`
[3] `https://global.flixbus.com/`

Chapter 2
Analysis

In this chapter, we will be looking into the current system of trying to buy sold-out tickets from two private providers, namely RegioJet and FlixBus. We will discuss their network communication and APIs used but also we will address the primary goals our system is trying to achieve and how are they being fulfilled as of now. We will also briefly discuss the target audience.

2.1 Target audience

As of the definition, we will treat the target audience of both providers as the same, considering their only difference is nationality(RegioJet is mostly focused on Czechia or Slovakia, FlixBus is more international), which we shall disregard.

Potential users include mostly people from the age range of fifteen to around fifty-five, both men and women, of all income levels. The age estimation varies among the users of public transportation, which includes both younger and older people. However, just a portion of the older would own and use a smartphone application to watch for tickets, and the younger usually do not have the financial means to buy the tickets, therefore leaving it to their parents.

2.2 Network communication

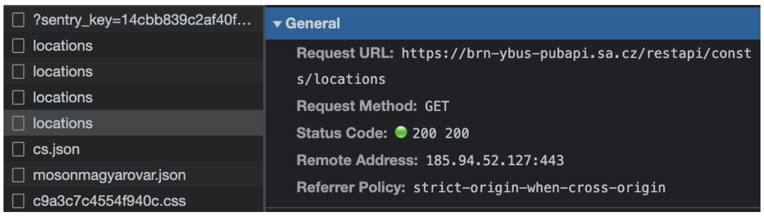
Both RegioJet and FlixBus systems use an API to communicate between their front-end and their back-end. This enables us to intercept this communication and analyze both requests and responses.

For the interception, we made use of the Chrome DevTools [1]. These tools include a network activity panel, that logs all network activity that happened while this panel was opened. These logs contain all the details about the activity, including but not limited to the request URL and method, headers of both response and request, and of course the payload of the request and the data returned by the response.

Using this information, we have all that we need to analyze the communication that happens during the process of finding and buying the tickets. The process is similar for both providers, and we will demonstrate it for RegioJet only, on a few examples.

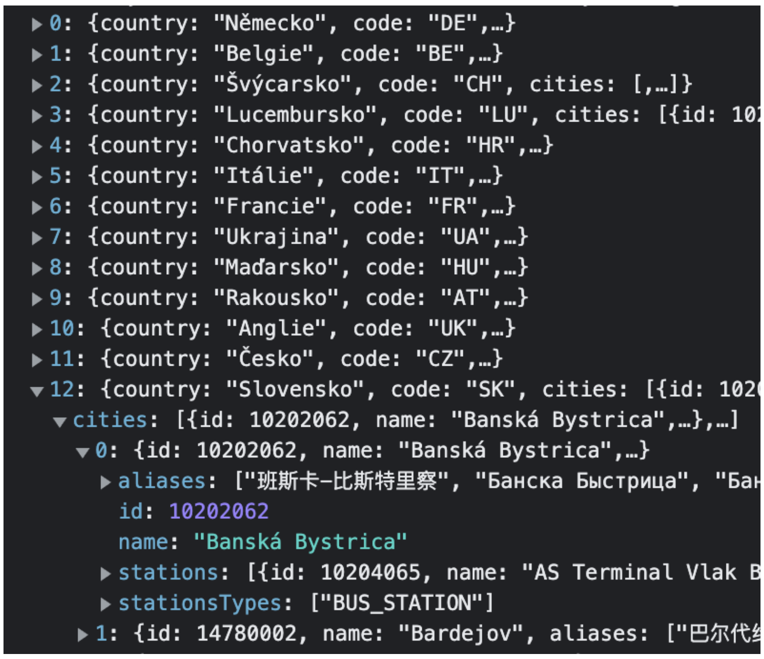
Firstly, we need to find the source of data for locations, such as Praha or Brno, and details about stations in these cities.

[1] `https://developer.chrome.com/docs/devtools/`



```
?sentry_key=14cbb839c2af40f…      ▾ General
locations                            Request URL: https://brn-ybus-pubapi.sa.cz/restapi/const
locations                            s/locations
locations                            Request Method: GET
locations                            Status Code: ● 200 200
cs.json                              Remote Address: 185.94.52.127:443
mosonmagyarovar.json                 Referrer Policy: strict-origin-when-cross-origin
c9a3c7c4554f940c.css
```

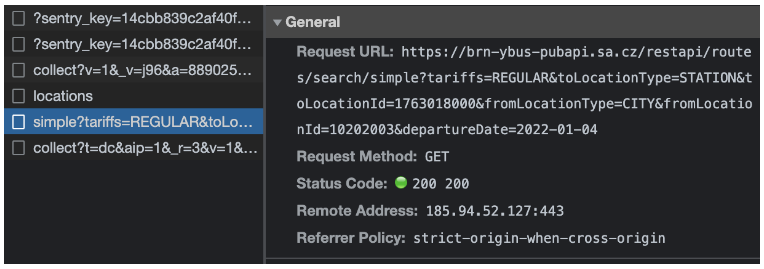
Figure 2.1. RegioJet request for locations



```
▶ 0: {country: "Německo", code: "DE",…}
▶ 1: {country: "Belgie", code: "BE",…}
▶ 2: {country: "Švýcarsko", code: "CH", cities: [,…]}
▶ 3: {country: "Lucembursko", code: "LU", cities: [{id: 10
▶ 4: {country: "Chorvatsko", code: "HR",…}
▶ 5: {country: "Itálie", code: "IT",…}
▶ 6: {country: "Francie", code: "FR",…}
▶ 7: {country: "Ukrajina", code: "UA",…}
▶ 8: {country: "Maďarsko", code: "HU",…}
▶ 9: {country: "Rakousko", code: "AT",…}
▶ 10: {country: "Anglie", code: "UK",…}
▶ 11: {country: "Česko", code: "CZ",…}
▼ 12: {country: "Slovensko", code: "SK", cities: [{id: 102
  ▼ cities: [{id: 10202062, name: "Banská Bystrica",…},…]
    ▼ 0: {id: 10202062, name: "Banská Bystrica",…}
      ▶ aliases: ["班斯卡-比斯特里察", "Банска Быстрица", "Бан
        id: 10202062
        name: "Banská Bystrica"
      ▶ stations: [{id: 10204065, name: "AS Terminal Vlak B
      ▶ stationsTypes: ["BUS_STATION"]
    ▶ 1: {id: 14780002, name: "Bardejov", aliases: ["巴尔代
```

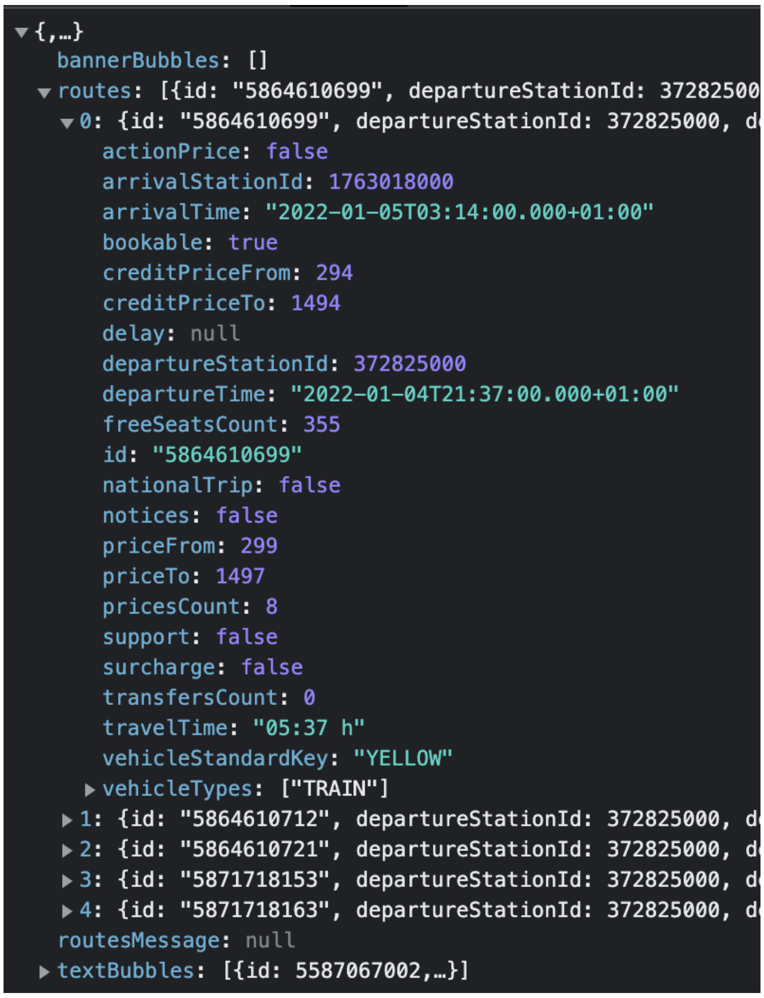
Figure 2.2. RegioJet response with locations

The second step involves investigating what payload is sent to which URL address in order to get the list of routes.



```
?sentry_key=14cbb839c2af40f…      ▾ General
?sentry_key=14cbb839c2af40f…         Request URL: https://brn-ybus-pubapi.sa.cz/restapi/route
collect?v=1&_v=j96&a=889025…         s/search/simple?tariffs=REGULAR&toLocationType=STATION&t
locations                            oLocationId=1763018000&fromLocationType=CITY&fromLocatio
simple?tariffs=REGULAR&toLo…         nId=10202003&departureDate=2022-01-04
collect?t=dc&aip=1&_r=3&v=1&…        Request Method: GET
                                     Status Code: ● 200 200
                                     Remote Address: 185.94.52.127:443
                                     Referrer Policy: strict-origin-when-cross-origin
```

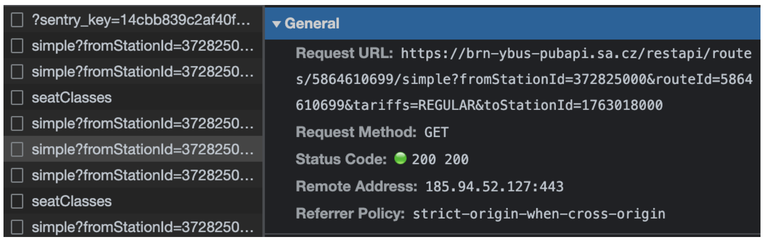
Figure 2.3. RegioJet request for routes



```
▼{,…}
   bannerBubbles: []
 ▼ routes: [{id: "5864610699", departureStationId: 37282500
   ▼0: {id: "5864610699", departureStationId: 372825000, d
      actionPrice: false
      arrivalStationId: 1763018000
      arrivalTime: "2022-01-05T03:14:00.000+01:00"
      bookable: true
      creditPriceFrom: 294
      creditPriceTo: 1494
      delay: null
      departureStationId: 372825000
      departureTime: "2022-01-04T21:37:00.000+01:00"
      freeSeatsCount: 355
      id: "5864610699"
      nationalTrip: false
      notices: false
      priceFrom: 299
      priceTo: 1497
      pricesCount: 8
      support: false
      surcharge: false
      transfersCount: 0
      travelTime: "05:37 h"
      vehicleStandardKey: "YELLOW"
     ▶vehicleTypes: ["TRAIN"]
   ▶1: {id: "5864610712", departureStationId: 372825000, d
   ▶2: {id: "5864610721", departureStationId: 372825000, d
   ▶3: {id: "5871718153", departureStationId: 372825000, d
   ▶4: {id: "5871718163", departureStationId: 372825000, d
   routesMessage: null
 ▶ textBubbles: [{id: 5587067002,…}]
```

Figure 2.4. RegioJet response with routes

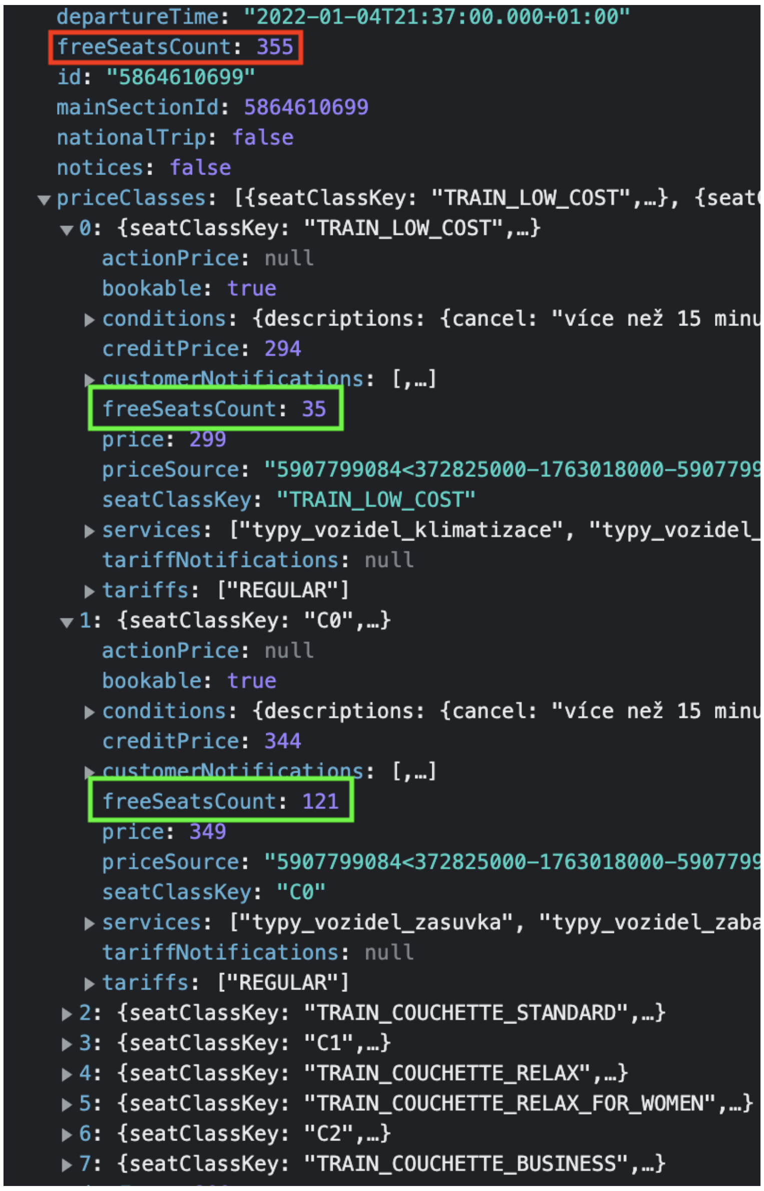
Lastly, we need to find out in which response we can see information about the availability of free seats.



```
?sentry_key=14cbb839c2af40f…
simple?fromStationId=3728250…
simple?fromStationId=3728250…
seatClasses
simple?fromStationId=3728250…
simple?fromStationId=3728250…
simple?fromStationId=3728250…
seatClasses
simple?fromStationId=3728250…
```

```
▼ General
   Request URL: https://brn-ybus-pubapi.sa.cz/restapi/route
   s/5864610699/simple?fromStationId=372825000&routeId=5864
   610699&tariffs=REGULAR&toStationId=1763018000
   Request Method: GET
   Status Code: ● 200 200
   Remote Address: 185.94.52.127:443
   Referrer Policy: strict-origin-when-cross-origin
```

Figure 2.5. RegioJet request for route's details



```
  departureTime: "2022-01-04T21:37:00.000+01:00"
  freeSeatsCount: 355
  id: "5864610699"
  mainSectionId: 5864610699
  nationalTrip: false
  notices: false
▼ priceClasses: [{seatClassKey: "TRAIN_LOW_COST",…}, {seat
  ▼ 0: {seatClassKey: "TRAIN_LOW_COST",…}
      actionPrice: null
      bookable: true
    ▶ conditions: {descriptions: {cancel: "více než 15 minu
      creditPrice: 294
    ▶ customerNotifications: [,…]
      freeSeatsCount: 35
      price: 299
      priceSource: "5907799084<372825000-1763018000-5907799
      seatClassKey: "TRAIN_LOW_COST"
    ▶ services: ["typy_vozidel_klimatizace", "typy_vozidel_
      tariffNotifications: null
    ▶ tariffs: ["REGULAR"]
  ▼ 1: {seatClassKey: "C0",…}
      actionPrice: null
      bookable: true
    ▶ conditions: {descriptions: {cancel: "více než 15 minu
      creditPrice: 344
    ▶ customerNotifications: [,…]
      freeSeatsCount: 121
      price: 349
      priceSource: "5907799084<372825000-1763018000-5907799
      seatClassKey: "C0"
    ▶ services: ["typy_vozidel_zasuvka", "typy_vozidel_zaba
      tariffNotifications: null
    ▶ tariffs: ["REGULAR"]
  ▶ 2: {seatClassKey: "TRAIN_COUCHETTE_STANDARD",…}
  ▶ 3: {seatClassKey: "C1",…}
  ▶ 4: {seatClassKey: "TRAIN_COUCHETTE_RELAX",…}
  ▶ 5: {seatClassKey: "TRAIN_COUCHETTE_RELAX_FOR_WOMEN",…}
  ▶ 6: {seatClassKey: "C2",…}
  ▶ 7: {seatClassKey: "TRAIN_COUCHETTE_BUSINESS",…}
```

Figure 2.6. RegioJet response with route's details. (Red are free seats overall, green are free seats in the given class)

This is a bit oversimplified, but these three steps are the core skeleton of actions needed to be taken for any-and-all providers that are to be supported by our system. In order for all of these steps to be finished, we may have to investigate more (or even all if needed) calls to APIs.

2.3 Current system

In the current system, the process of reserving sold-out tickets is either time-consuming or based purely on luck. A user searches for the route he wants and checks if any tickets are available. If there are none, he waits some amount of time, based on the urgency. If the transport leaves tomorrow, he may be checking the site once every 10 minutes, but if it leaves next week, he can check only once per day. The next step is for him to hope somebody canceled their ticket, and what is more, that no one got there before him and booked the ticket already.

2.4 Goals

The system described in this thesis will address and will get rid of the issues of both time consumption and of random success rate.

Users will only have to search for the route once, even if the tickets are sold out, and the repetitive checks of availability will be carried out quickly and periodically by our system using analysis of the API responses. By using a mobile application, we can ensure that the user will know when he will be able to book the tickets, as the moment they become available push notifications will be sent to his mobile device.

2.5 Components

Based on the aforementioned goals that we laid out, we know our system will consist of two components. We can think of it as parts of the web systems. There is a back-end side and a front-end side of the system. The backend part of the application sometimes called „the server-side“ is basically how the site works, updates, and changes. This refers to everything the user cannot see, like databases and servers. On the backend, we manipulate and store all the data, like user profiles, images, etc. The front-end is everything involved with what the user sees and interacts with, including design.

Front-end will be the mobile application that will provide an interface for users to search and select which routes they want to watch. It will be pure UI, with only the responsibility to show and send data from and to the back-end.

The server side will be a set of public APIs. By providing well-designed end-points for the UI to use, we can manage the application using simple calls. The whole logic will reside in this system. Its responsibility includes emulating APIs of other providers and parsing their responses to get the data for all locations and stations, and all routes between them. Furthermore, it will store the watched routes of each and every user, and periodically check the availability of the tickets, and finally, it will create a push notification that will be sent to the user's device. We will discuss it in the next chapter.

2.6 Conclusion

This chapter aims to explain the process of analysis that is needed to understand how one can emulate their system to automatically detect the change in the availability of free seats. Moreover, it describes the current pain points of trying to buy sold-out tickets and proposes an outline of a solution. We will build on this in the later chapters.

Chapter 3
Server-side

In this chapter, we will be looking into the server-side, or back-end, of the proposed system. We will discuss the reasoning behind the technologies that were chosen and behind the overall architecture. We will build upon this with an overview of implementation in the next chapter.

3.1 Technology

For the language of the server, we decided to go with Java[1]. One of the reasons is a familiarity with the language, but most importantly, with Java, we can use the framework Spring[2].

Spring is a framework that „... provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform. A key element of Spring is infrastructural support at the application level: Spring focuses on the `plumbing` of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments.“[1]

These are not just marketing words. With Spring, one can focus mostly on the application logic itself, and write code, without worrying about the underlying environment. Moreover, „at its core, Spring Framework's Inversion of Control (IoC) and Dependency Injection (DI) features provide the foundation for a wide-ranging set of features and functionality.“[2]

These features will be explained in more detail in the Implementation 4 chapter.

Using Java and Spring, the best tool to provide data for a mobile application is via Web API[3] which the application will be communicating with.

3.2 Database

For storing information we need a database. Nowadays there are two main options:

- Relational database (often mentioned as „SQL databases“)
- Non-relational database (often mentioned as „NoSQL databases“)

These two categories may seem contradictory to each other. In fact, they are. To some degree. To best annotate their difference, we can compare them to hand tools. A saw and a knife may do the same job - cutting -, but one of them is better for cutting soft materials whilst the other one is better for hard materials. This does not mean one is better than the other. The same is true for the databases. „When comparing relational and non-relational databases, it's important to first note that these two very

[1] `https://www.java.com/en/`
[2] `https://spring.io/`
[3] `https://www.w3schools.com/js/js_api_intro.asp`

different types of databases are equally useful in their own right—but for contrasting reasons and use-cases. One type of database is not better than the other type, and both relational and non-relational databases have their place. “[3]

To plainly state the main differences between these two categories: „..relational databases store data in rows and columns like a spreadsheet while non-relational databases store data don’t, using a storage model (one of four) that is best suited for the type of data it’s storing. “[4]

In the end, we chose to go with the NoSQL database, in particular MongoDB[1]. This decision was taken because the data we will be dealing with does not have relations. For example, we will be storing the details of watched routes, such as the id of the route, arrival and departure time, and seat classes that we should watch. This usage resembles documents and is, therefore, better suited for NoSQL databases. Another reason for choosing MongoDB is its easy and intuitive integration with the Spring framework.

3.3 Architecture

We will be building a REST API. „A REST API is an application programming interface (API) that uses a representational state transfer (REST) architectural style. The REST architectural style uses HTTP to request access and use data. This allows for interaction with RESTful web services.“[5]

REST works on top of the HTTP. It takes advantage of its native capabilities, such as GET, PUT, POST, and DELETE. When a request is sent to a RESTful API, the response returns in one of the few formats, such as either JSON, XML, or HTML.

A RESTful API is defined by a web address, or Uniform Resource Identifier (URI), typically following a naming convention. For an API to be considered RESTful, it has to conform to these criteria[6]:

- A client-server architecture
- Stateless client-server communication
- Cacheable data
- A layered system
- A uniform interface between components. This requires that:
  - Resource identification in requests
  - Resource manipulation through representations
  - self-descriptive messages
  - Hypermedia as the engine of application state

In the next chapter, we will show how our implementation conforms to these criteria.

Finally, the goal of our Web API is to provide all the data from providers and emulate their behavior. As the main steps are similar between providers, and we may want to add another one in the future, the main part of the design is to be as generic as possible, to remove the redundancy and increase the modularity of our system.

3.4 Conclusion

This chapter aims to explain the basis of the technology chosen for the project and the used architecture. It mentions an overview of REST API with its methods. Moreover, it describes the criteria for an API to be considered RESTful. We will build upon this with an overview of our implementation in the next chapter.

[1] `https://www.mongodb.com/`

Chapter 4
Implementation

In this part, we will go through a high-level overview of the implementation of the application's back-end, including the implementation and structure of the database and the data in it.

4.1 Design

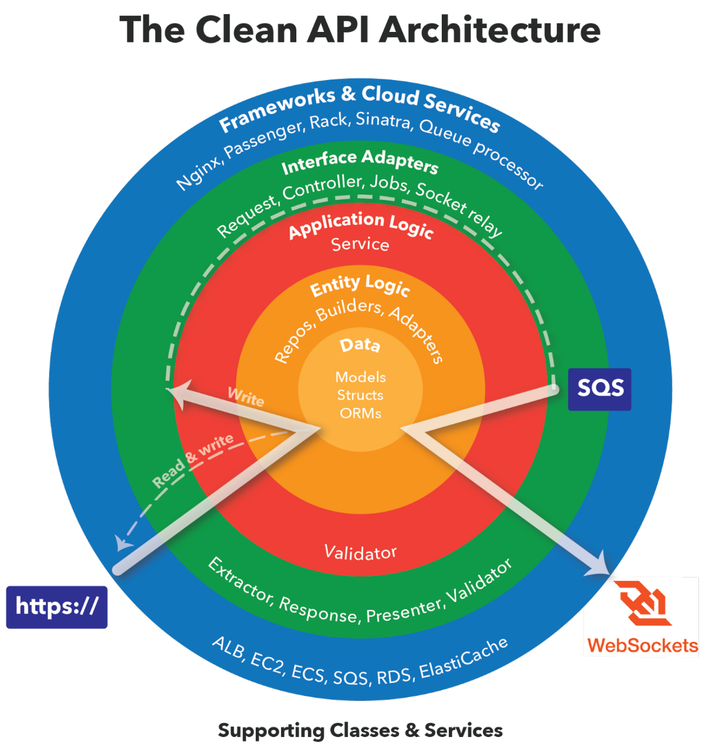

The Clean API Architecture
Frameworks & Cloud Services
Nginx, Passenger, Rack, Sinatra, Queue processor
Interface Adapters
Request, Controller, Jobs, Socket relay
Application Logic
Service
Entity Logic
Repos, Builders, Adapters
Data
Models
Structs
ORMs
Write
Read & write
SQS
Validator
Extractor, Response, Presenter, Validator
ALB, EC2, ECS, SQS, RDS, ElastiCache
https://
WebSockets
Supporting Classes & Services

Figure 4.1. Layered API Architecture[7]

For the first layer, we are using the aforementioned Spring Framework, which manages incoming requests and traffic. These requests are routed to corresponding controllers[1].

To fully integrate layers into our architecture, we had to make sure that their purpose is only to extract each parameter, validate its syntax, and authenticate the user making the request, but most of the logic is done in the Application logic layer. Therefore, our controllers only pass the arguments into the next layer and return the results of that code in the right format. Example of one API endpoint controller code:

[1] `https://zetcode.com/springboot/restcontroller/`

```
@GetMapping(value = "/routes")
public RoutesDto getRoutes(
 @RequestParam String tariffs,@RequestParam String toLocationType,
 @RequestParam String toLocationId, @RequestParam String fromType,
 @RequestParam String fromLocationId, @RequestParam String departureDate)
 {
return service.getRoutes(tariffs, toLocationType, toLocationId,
fromLocationType, fromLocationId, departureDate);
 }
```

Every controller has at least one service containing the application logic. This layer makes sure the user is authorized to access data, and then retrieves data from the next layer.

The Entity Logic Layer can work through a Repository for databases and through an Adapter for APIs. In our system, both are used. Generic Adapter is used to get data from providers' APIs, and Mongo Repository is used for data that we store.

The last layer is a Data Layer, which ideally should be really simple, only containing the interface for the storage, like Database Entity classes, or other models.

4.2 Code

4.2.1 Common code base

Firstly, we needed to convert JSON responses from providers' APIs into a Java objects. For this, we used a web service JSON2CSharp[1] that converts JSON objects into Plain Old Java Objects(POJOs). For example this is a model for RegioJet's delay response:

```
public class Delay {
    private String busConnectionId;
    private String label;
    private String number;
    private int delay;
    private String vehicleCategory;
    private int freeSeatsCount;
    private List<ConnectionStation> connectionStations;
}
```

As you can see, it contains `ConnectionStation`, another model. This means that one response can be made up of multiple models.

Using this, we went on and created a model for every RegioJet API response. The next step was to create Data Transfer Objects(DTOs) from these models. DTO „is an object that carries data between processes. The motivation for its use is that communication between processes is usually done by resorting to remote interfaces (e.g., web services), where each call is an expensive operation."[8] As response data contains a lot of information, and not all of it relevant to our usage, we can reduce the number of data send. Compare the model and the DTO for a RegioJet's station:

```
public class Station {
    private String id;
    private String name;
```

[1] https://json2csharp.com/json-to-pojo

```
    private String fullname;
    private List<String> aliases;
    private String address;
    private List<String> stationsTypes;
    private String iataCode;
    private String stationUrl;
    private String stationTimeZoneCode;
    private String wheelChairPlatform;
    private int significance;
    private double longitude;
    private double latitude;
    private String imageUrl;
}
```

```
public class StationDto {
    private String id;
    private String name;
    private String fullname;
}
```

To transform the model to DTO(and vice versa), we needed a Mapper class. As we mentioned before, to speed up the development process and limit redundancy, we tried to use generics[9] everywhere where it made sense. Therefore, we created an abstract class that already contains the mapping logic for lists, and to have it work with each and any model/DTO duo, one just has to inherit from this class and provide an implementation for an abstract method.

```
public abstract class ToDtoMapper<Model, Dto> {

    /**
     * Maps model to dto
     *
     * @param models list of models
     * @return list of dtos
     */
    public List<Dto> mapToDto(Collection<Model> models) {
        final List<Dto> result;
        if (models != null) {
            result = models
                .stream()
                .filter(Objects::nonNull)
                .map(this::mapToDto)
                .collect(Collectors.toList());
        } else {
            result = List.of();
        }
        return result;
    }

    protected abstract Dto mapToDto(Model model);
}
```

And the Mapper for Station model and its DTO that were shown above.

```
public class StationDtoMapper extends ToDtoMapper<Station, StationDto> {

    @Override
    public StationDto mapToDto(Station station) {
        if (station == null) {
            return StationDto.builder().build();
        }
        return StationDto.builder()
            .id(station.getId())
            .name(station.getName())
            .fullname(station.getFullname())
            .build();
    }
}
```

After we repeated this for every model and DTO, we moved on to another part of the system. This time, we focused on everything that could be called „common“ for all providers.

The most obvious part was an adapter to get data from given APIs. For every provider, the URL and even return type would be different, but the basic functionality stays the same. Fetch data from this URL and return them as the correct Java model. To fulfill this, generics are needed again.

```
public class ApiAdapter<U extends ApiClient> {

    private final U client;

    public <T> T get(String uri, HttpEntity<?> requestEntity,
    ParameterizedTypeReference<T> responseType) {
        T response = client
         .callApi(uri, HttpMethod.GET, requestEntity, responseType);
        return response;
    }
    public <T> T post(String uri, HttpEntity<?> requestEntity,
    ParameterizedTypeReference<T> responseType) {
        T response = client
         .callApi(uri, HttpMethod.POST, requestEntity, responseType);
        return response;
    }
    public <T> T put(String uri, HttpEntity<?> requestEntity,
    ParameterizedTypeReference<T> responseType) {
        T response = client
         .callApi(uri, HttpMethod.PUT, requestEntity, responseType);
        return response;
    }
    public <T> T delete(String uri, HttpEntity<?> requestEntity,
    ParameterizedTypeReference<T> responseType) {
        T response = client
         .callApi(uri, HttpMethod.DELETE, requestEntity, responseType);
        return response;
    }
}
```

Where U is of type that extends `ApiClient`. `ApiClient` provides an interface to call given API and return always the correct model, based on the `ParameterizedTypeReference`.

```
public interface ApiClient {
    default <T> T callApi(String uri, HttpMethod method,
     ParameterizedTypeReference<T> responseType) {
        return callApi(uri, method, null, responseType);
    }

    <T> T callApi(String uri, HttpMethod method, HttpEntity<?> entity,
     ParameterizedTypeReference<T> responseType);
}
```

This way, we can easily swap clients, without any change in the code except for one word. We went in this direction because we wanted the system to be as modular as possible. For example, if we wanted to mock responses, we may create a `MockClient` that would return statically created data. However, an even better use case showed itself during the development. From the beginning, we were using Spring's RestTemplate[1] class. After finishing every functionality for RegioJet, we found out that a new and better client was available, namely WebClient[2]. Thankfully, by building the ApiAdapter using generics and thanks to Spring's Dependency Injection, we only had to create a new class implementing `ApiClient`. Let's see both clients first, and then we will show how easily we can swap them.

```
public class RestTemplateClient extends ApiClientWithExceptionHandling {
    private final RestTemplate restTemplate;
    @Override
    protected <T> T callApiImplementation(String uri, HttpMethod method,
     HttpEntity<?> requestEntity, ParameterizedTypeReference<T> type) {
        ResponseEntity<T> rateResponse =
            restTemplate.exchange(uri,
                method, requestEntity, type);
        return rateResponse.getBody();
    }
}
```

```
public class WebFluxClient extends ApiClientWithExceptionHandling {
    private final WebClient webClient;

    @Override
    protected <T> T callApiImplementation(String uri, HttpMethod method,
     HttpEntity<?> requestEntity, ParameterizedTypeReference<T> type) {

        Mono<T> response = webClient
            .method(method)
            .uri(URI.create(uri))
            .retrieve()
            .bodyToMono(type);
        return response.block();
    }
```

[1] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html
[2] https://www.baeldung.com/spring-5-webclient

As we can see, just by extending an ApiClientWithExceptionHandling - which is a class building upon `ApiClient`, but enforcing logging and error handling - and overriding one method, we have a totally different client to work with APIs, without changing any of the other classes. To swap between RestTemplate and WebFluxClient (and vice versa), we only need to do the following change:

```
private final ApiAdapter<RestTemplateClient> client;
```

```
private final ApiAdapter<WebFluxClient> client;
```

No other change in code is needed, yet under the hood everything is different.

4.2.2 Provider's specifics

Even with the focus on generics, every provider has to have some specific classes, like its own controller and service that emulate RegioJet's API.

For example, one endpoint of RegioJet controller:

```
@RequestMapping("/api/v1/regio")
public class RegioJetController {

    private final RegioService service;
    private final WatchedRegioConnectionDao watchDao;

    @GetMapping(value = "/locations")
    public List<LocationDto> getLocations() {
        return service.getLocations();
    }
...
...
}
```

Now, calling address `.../api/v1/regio/locations` will return all the data we selected about locations from RegioJet's API. As we can see, the endpoint is simply calling a RegioService method and returning a list of DTOs. So let us take a look at the service.

```
public class RegioService implements ServiceWithHealth {
    private final ApiAdapter<WebFluxClient> client;
    private final RegioMapper mapper;

    public List<LocationDto> getLocations() {
        List<Location> locations = client
         .get(YbusApi.getLocationsUrl(),
new ParameterizedTypeReference<>() {
        });
        return mapper.mapToDto(locations);
    }
...
...
}
```

What is going on here is very simple. We get the URL for delay's API from `YbusApi` class that stores all the information about RegioJet's URLs used in APIs. We pass it into the ApiAdapter and use GET method. This returns the data that RegioJet's API has. The last step is to map this data to a list of DTOs, so we can reduce the size of the transferred data.

This is similarly done for all remaining endpoints.

4.2.3 Caching

The attentive reader may have noticed that this system can cause unnecessary stress on the provider's server. A list of locations is not something that changes often, same as for example a list of tariffs or seats.

One possible solution is to store this data on our side in the database. This comes with its own set of problems. The first is an increase in the storage space, which may increase the cost of the server, depending on the provider. The second is the sync up of the local data with the source data. Should we check every time if we have all the data, or if some items have been changed? Another approach would be to have a scheduled task that synchronizes the data after a predefined period of time.

This is an approach that would make sense and would work, but the implementation would take us some time. Therefore we decided to go with a solution that is in principle the same, yet different. The aforementioned problem is one of the most common problems in API development, and there is a solution with tested and optimized implementations. This solution is called caching.

„Caching is a mechanism to improve the performance of any type of application. Technically, caching is the process of storing and accessing data from a cache. But wait, what is a cache? A cache is a software or hardware component aimed at storing data so that future requests for the same data can be served faster.“[10]

The last sentence summarizes the value of caching. We do it so we can quickly serve requests after the first initial one. For example, only the first user getting the locations will have to wait for the data to be fetched from the provider. All subsequent queries will use this first response automatically. No need to call the provider's API or query the database.

This is where the initial choice of language and framework comes into play. As stated in the Spring documentation: „The Spring Framework provides support for transparently adding caching to an application. At its core, the abstraction applies caching to methods, thus reducing the number of executions based on the information available in the cache. The caching logic is applied transparently, without any interference to the invoker. Spring Boot auto-configures the cache infrastructure...“[11]

What does this mean? Basically, we only need to add an annotation to the method we want to cache. Let us show it on an example:

```
@Cacheable("tariffs")
public List<TariffDto> getTariffs(
@RequestParam(defaultValue = "cs") String language,
@RequestParam(defaultValue = "CZK") String currency) {
    setVariables(language, currency);
    return service.getTariffs();
}
```

As we can see, we have annotation @Cacheable. Its parameter is the name of the cache that should be created. Without specifying a look-up key, all the parameters will be combined to create one key. Now, all but the first invocation of the method getTariffs with the same arguments will use stored response.

However, this is only an abstraction, meaning, this does not provide us with the real implementation. The cache itself could be done as a database, on disk, or in memory.

We must choose the implementation that would fit our needs the best. Fortunately, with Spring, there are many options, such as Encache, Guave, Redis, or Caffeine.

As we only need to store the data while the server is running, we need just an in-memory cache, with no disk persistence. Our second need is to have two different lifespans for the cache. The first is data that is not supposed to be changed much, or at all. Tariffs, seats, and locations fall into this category. For them, we can have a cache with a long time to live(TTL), such as a few days, or - as we selected- 6 hours. The second category is a bit more specific. All the other calls, such as a list of routes or connection details, belong to it. We want to store the responses for a short time - for three minutes in our case-, as the resulting data can be changed quickly. So why cache this data at all? Imagine two users searching for routes from Prague to Brno at almost the same time. Without cache, we would need to fetch the data from the provider's API two times, and this can be slow. However, when we cache the first result for a few minutes, we call it only once and then return this data for the second user as well. With the TTL so short, we can be reasonably sure the data closely resembles the correct data.

From the cache options, the best fit for our needs is Caffeine[1]. As stated in their documentation, caffeine „ is a high performance Java caching library providing a near optimal hit rate. A Cache is similar to ConcurrentMap, but not quite the same. The most fundamental difference is that a ConcurrentMap persists all elements that are added to it until they are explicitly removed. A Cache on the other hand is generally configured to evict entries automatically, in order to constrain its memory footprint. “[12]

As it is explained above, the Caffeine cache is a special implementation of Java's ConcurrentMap[2], therefore the cache lives in memory, as we wanted. The inner workings are quite simple, a key is created from all the method's parameters, and the method's response is stored with this key in the map for a specified amount of time.

Now, to enable the cache in our application, we just need to create cache managers for the two aforementioned categories. With Spring, this is as easy as creating two beans:

```
@Bean
@Primary
public CacheManager cacheLongManager() {
    CaffeineCacheManager manager = new CaffeineCacheManager();
    caffeineCacheManager.setCaffeine(Caffeine
            .newBuilder()
            .expireAfterWrite(6, TimeUnit.HOURS));
    return manager;
}

@Bean
public CacheManager cacheShortManager() {
    CaffeineCacheManager manager = new CaffeineCacheManager();
    caffeineCacheManager.setCaffeine(Caffeine
            .newBuilder()
            .expireAfterWrite(3, TimeUnit.MINUTES));
    return manager;
}
```

[1] https://github.com/ben-manes/caffeines
[2] https://www.baeldung.com/java-concurrent-map

The only thing left is to specify the correct manager for the correct caches. This is done by providing a cacheManger parameter to the annotation.

```
    @Cacheable(cacheNames="route", cacheManager = "cacheShortManager")
    public ConnectionDto getConnection
```

After these steps, all responses from methods annotated with @Cacheable are automatically stored and reused when the method is called with the same parameters, improving the efficiency and speed of the server.

4.2.4 RESTful API

In the previous chapter 3.3 we noted that for the API to be considered RESTful, it has to conform to some criteria. Here we will argue that this system is a RESTful one.

- A client-server architecture - our system follows Separation of concerns[13], separating the user interface from the data storage
- Stateless client-server communication - no session information is stored nor needed in our system. Every call can be understood in isolation, as it contains all the necessary information.
- Cacheable data - some of the data cannot be cached (for longer than a few minutes), e.g. the route's details with the number of available seats, but most of the data is supposed to be cached, for example, the cities and stations will not be changed frequently. See section 4.2.3 above for more details.
- A layered system - our system does not care whether it is connected directly to the end server or to a proxy along the way, its communication is not dependent on the recipient.
- A uniform interface between components - individual resources are identified in requests, and the resources themselves are separate from their internal representations, e.g in the database, it is stored as long int, but in the JSON response string may be returned.

4.2.5 Database

Following is the high-level overview of the implementation of the database mentioned in 3.2. Spring has - as with most of the libraries and technologies - nice and easy integration with MongoDB. Using MongoTemplate and MongoRepository, the whole integration can be done in a few lines of code.

„MongoTemplate — MongoTemplate implements a set of ready-to-use APIs. A good choice for operations like updates, aggregations, and others, MongoTemplate offers finer control over custom queries. MongoRepository — MongoRepository is used for basic queries that involve all or many fields of the document. Examples include data creation, viewing documents, and more.“[14]

There are only two steps that are required to be taken in order for the Spring application to work with MongoDB. The first step is to create an entity object. This entity will specify what we want to store, as well as the naming, both the collection and the stored objects. The second step consists of creating an interface for a repository, and optionally creating method definitions on this interface. Let us show it on some examples. Firstly, for the entity object, we will explain the details on our WatchedRegioConnectionEntity - object with all the information we need to watch a specific route.

```
@Getter
@Setter
@Document(collection = "regio.watched_connections")
public class WatchedRegioConnectionEntity {
    @Id
    private String id;
    @NotBlank
    private String userId;
    @NotBlank
    private String url;
    @NotBlank
    private String language;
    @NotBlank
    private String routeId;
    @NotBlank
    private String fromStationId;
    @NotBlank
    private String toStationId;
    @Min(1)
    private int tickets = 1;
    @NotNull
    private List<String> tariffs;
    private List<String> seatClasses;
    private Map<String, Date> notified = new HashMap<>();
    @NotNull
    @JsonFormat(pattern="yyyy-MM-dd HH:mm:ss.SSSX")
    private Date arrivalTime;
    @NotNull
    @JsonFormat(pattern="yyyy-MM-dd HH:mm:ss.SSSX")
    private Date departureTime;
    @NotBlank
    private String type;
}
```

The first thing that may have caught your attention is the annotations. There are four different kinds of them. Firstly, there are Lombok[1]. annotations. These annotations serve as code generation blocks. In our code, we are using the most common annotations, @Getter and @Setter. Thanks to them, code for getters and setters is created automatically during the compilation. Another example could be annotation @Builder, which will create a builder for your class, or @RequiredArgumentsConstructor (and his parallels of No and All arguments), which will create a constructor for the required arguments.

Secondly, there are MongoDB annotations, namely @Document and @Id. These annotations are used as configuration tools. Within the document annotation, we specify the name of the collection in which this entity should be stored. For @Id, it does exactly what its name hints - it marks the attribute as a primary index.

Thirdly, we have JavaX validation annotations. These serve as validators, making sure the data inside the entity comply with our rules. For instance, the @NotBlank annotation makes sure that the string value is not null or whitespace, @Min specifies the minimum permitted value inside the integer field, and so on. These rules are checked

[1] https://projectlombok.org/

when we try to save the entity into the database, and if any of them fails, the saving fails as well.

Lastly, we have a @JsonFormat annotation from Jackson[1]. With this annotation, we can control the output format of Date data types.

Now, we can take a look at the actual data. To watch over a connection, we need all the data specified in the entity. Let us take a look at each of them:

- id - autogenerated unique id for a database entity
- userId - unique id of the user's device generated by Firebase (will be explained in chapter 5)
- language - language in which the notification to mobile should be sent
- routeId - id of the route
- fromStationId - id of the departure station
- toStationId - id of the arrival station
- tickets - number of tickets desired
- tariffs - list of tariffs specified by the user
- seatClasses - list of seat classes which we want to watch
- notified - map with string keys and datetime values, representing the time when the user has been notified from watcher of type key
- arrivalTime - time of the supposed arrival
- departureTime - time of the supposed departure
- type - the type of the watcher that should be used, can be combined, e.g. string „ticketsdelays“ is used when watching for both tickets and delays,

This is all that we need to actually watch for changes in the connection, as we will show in the next chapter.

The second step of implementing MongoDB inside the Spring framework is easier and quicker, but even more powerful. We just have to create a new interface that is extending the already defined MongoRepository interface.

```
@Repository
public interface WatchedRegioConnectionRepository
extends MongoRepository<WatchedRegioConnectionEntity, String> {
    List<WatchedRegioConnectionEntity> findByTypeLike(String type);
}
```

As you can see, we also created a new method declaration, but this step is optional. All that is required to have basic functionality is to create an empty interface, and after that, methods such as findAll, save, saveAll are available for you to use.

However, the option to create new methods is a really powerful one, due to how Spring can create a method for querying the database automatically from the name of the method, so you do not have to create your own queries. Let us explain it in our example:

```
List<WatchedRegioConnectionEntity> findByTypeLike(String type);
```

The return type should usually be either a list of your entities or a single entity. The name complies with the following pattern:

```
findBy<fieldName><method>
```

and it takes an argument(s) to this method.

Methods are similar to those used in SQL. Few examples:

[1] https://github.com/FasterXML/jackson

```
findByAgeGreaterThan(int age),
findByAgeBetween(int from, int to)
findByFirstnameNull()
findByFirstname(String name)
findByActiveIsTrue()
```

[15]

4.2.6 Watchers

The most important part of this back-end (except the API emulation) system is „watchers“. These are the scheduled actions that run periodically and check whether the watched condition evaluates to true. Right now, for RegioJet, watchers for the availability of tickets, and delay checks are implemented. For FlixBus, both of the aforementioned watchers are created, as well as another one that checks whether the connection has not been canceled. However, this is a topic bigger in scope, and as such we will discuss it in chapter 5.

4.3 Conclusion

This chapter aims to explain the high overview of the implementation and its design. It provides code examples to illustrate the discussed points. An explanation and reasons are provided for the caching mechanism used in this implementation. Moreover, it describes why the implementation conforms criteria for a RESTful API. The structure of the database and the data in it is described, alongside its implementation in the Spring framework. This chapter mentions watchers which will be an important part of the discussion in the next chapter. The whole code is available on Github `https://github.com/alim971/watchdog/tree/rc`.

Chapter 5
Watchers

In the following section, we will talk about watchers - the scheduled tasks that are responsible for periodic checking of watched connections. We will explain the architectural overview of their structure, and provide examples of their workings. This will also require us to talk about the schedule functionality of the Spring framework, which we will quickly cover. In the end, we will talk about our current functioning watchers.

5.1 Overview

The main bulk of the back-end service is its ability to detect changes on the watched route. This is where the watchers come into the play. They use data from the database to periodically monitor responses from the provider's API. In case a wanted requirement is met, the watcher notifies the user that saved this connection. The notification is sent via Firebase Cloud Messaging. „Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably send messages at no cost.“[16]

If a notification was sent, another one for the same type cannot be sent again for a specified amount of time. Currently, this time is set to ten minutes, but in the future, this can be possibly replaced by users' custom values.

If the conditions are met again after the aforementioned period, the notification is sent again and the whole process repeats.

5.2 Scheduled

The watcher's job is to periodically check the API responses. That is why we need to repeatedly run the watcher at some interval. There are multiple approaches to this problem. For example, one could create a script and then using cron execute it periodically. However, this would only add another layer and complexity to our application. We wanted to avoid that. Thankfully, Spring has this functionality built-in.

We start by enabling the Scheduling tasks using the annotation @EnableScheduling on our main application class. With this done, all we have to do is create the task that should be run periodically. We do this by creating a function (preferably with a return type of void, as anything returned will be ignored) and annotating it with @Scheduled.

```
@Scheduled(fixedDelay = 10000)
    public void watch() {
        watch(getWatchedConnections());
    }
```

As you can see above, we have declared a method called watch, which is annotated as a Scheduled task. We also provided the parameter fixedDelay which sets the schedule of execution at 10000 milliseconds after the previous execution. This way, the task always waits until the previous one is finished.

There are other options than fixedDelay. We could use fixedRate, with the difference being that fixedRate executes the function every n milliseconds, even if the previous execution has not ended. Instead of delays and rates we could also use the flexibility of cron expression to control the executions. Let us show it on example:

```
@Scheduled(cron = "0 15 10 15 * ?")
    public void watch() {
        watch(getWatchedConnections());
    }
```

In this example, the function will be executed at 10:15 on the 15th day of every month. For more information about cron expressions see `https://docs.oracle.com/cd/E12058_01/doc/doc.1014/e12030/cron_expressions.htm`.

5.3 Structure

As mentioned in the previous chapters, during implementation we placed importance on the ease of extendability by using generics. Implementation of watchers is no exception. Firstly, we created an abstract Watchdog class that has the implementation of the tasks dealing with notifications and database access.

With the database access in mind, in the previous chapter, we described the data we store about the watched route. Watchers use this data so they can get details from the provider's API. Therefore, the data is dependent on the exact provider, but some attributes must be the same for all of them. This data is abstracted in the Notifiable interface:

```
public interface Notifiable {
    Map<String, Date> getNotified();
    String getUserId();
    String getLanguage();

    void setNotified(Map<String, Date>  notified);
}
```

The entities must implement this interface, making sure that we have the data we need to notify the correct user - userId and language of the notification. Also, we need to have a Map of string keys and date values that will be used to set the time when the notification was sent to the user for the specific type of watcher, to avoid repeatedly sending the same notification.

Now we ensured that we have all the needed data that the abstract watchdog will use. Let us see the implementation:

```
@Service
@RequiredArgsConstructor
public abstract class Watchdog<Model extends Notifiable, Entity> {
    protected final WatchedDao<Model, Entity> watchDao;
    protected final NotificationSenderService senderService;
    protected final String type;

    @Scheduled(fixedDelay = 10000)
    public void watch() {
        watch(getWatchedConnections());
    }
    protected abstract void watch(List<Model> connections);

    protected List<Model> getWatchedConnections() {
        return watchDao.getAllWatched();
    }

    protected void notifyUser(
    Model connection, Map<String, String> data, String message)
     throws FirebaseMessagingException {
        System.out.println(message);
        NotificationWithData notification = NotificationWithData
                .builder()
                .title(TranslationService
                    .getTitleTranslation(connection.getLanguage(), type))
                .message(message)
                .data(data)
                .build();
        senderService.sendNotification(notification,
              connection.getUserId());
        setNotifyTimeToNow(connection);
    }

    protected void notifyUser(Model connection)
    throws FirebaseMessagingException {
        notifyUser(connection, null,
        TranslationService
        .getFoundTicketsTranslation(connection.getLanguage()));
    }

    protected final void setNotifyTimeToNow(Model connection) {
        Map<String, Date> notified = connection.getNotified();
        notified.put(type, new Date());
        connection.setNotified(notified);
        watchDao.saveModel(connection);
    }

    protected final boolean wasNotified(Notifiable connection) {
        int timeToAdd = 10 * 60 * 1000;
        return wasNotified(connection, timeToAdd);
    }

    protected final boolean wasNotified(
    Notifiable connection, int timeToAdd) {
        Date now = new Date();
        return connection.getNotified() != null
            &&
            connection.getNotified().containsKey(type)
            &&
            now.before(new Date(
            connection.getNotified().get(type).getTime() + timeToAdd));
    }
}
```

Now, as you can see, everything is implemented already, there is only one abstract method without implementation. This method is the one called periodically, it is the one with the whole logic and watched condition. The concrete watcher needs to implement only this method and the method to get the saved connections with the correct watcher type.

```
    protected abstract void watch(List<Model> connections);

    protected List<Model> getWatchedConnections() {
        return watchDao.getAllWatched();
    }
```

There is one more intermediate step. For every provider, we create one more abstract class that will extend the abstract Watchdog with the correct models and entities and provide the corresponding service that will be handling API calls.

```
public abstract class RegioWatchdog
extends Watchdog<WatchedRegioConnection, WatchedRegioConnectionEntity> {

    protected final RegioService service;

    public RegioWatchdog(WatchedRegioConnectionDao watchDao,
     NotificationSenderService sender,
    String type, RegioService service) {
        super(watchDao, sender, type);
        this.service = service;
    }
}
```

5.4 Examples

For now, we have watchers for tickets and delays for both RegioJet and FlixBus. The latter has one additional watcher to check whether the route has been canceled.

We will show you two examples of the watchers to further explain the inner workings and the ease with which a new watcher can be added.

Let us start with the watcher for RegioJet tickets availability:

```
@Service
@Slf4j
public class TicketsWatchdog extends RegioWatchdog {
    public TicketsWatchdog(WatchedRegioConnectionDao watchDao,
     NotificationSenderService sender, RegioService service) {
        super(watchDao, sender, "tickets", service);
    }

    @Override
    protected void watch(List<WatchedRegioConnection> connections) {
        for (WatchedRegioConnection connection : connections) {
            try {
                if (wasNotified(connection)) {
                    continue;
                }
                if (connection.getArrivalTime().before(new Date())) {
                    throw new ClientException(
                "Connection is already finished");
        }
        ConnectionDto result = service.getConnection(
            connection.getTariffs().get(0),
            connection.getRouteId(),
            connection.getFromStationId(),
            connection.getToStationId()
        );

        long routesSatisfiable = connection
             .getSeatClasses() != null
        && !connection.getSeatClasses().isEmpty()
            ? result.getPriceClasses().stream()
                    .filter(Objects::nonNull)
                    .filter(e -> connection
                    .getSeatClasses()
                    .contains(e.getSeatClassKey())
                     &&
                     e.getFreeSeatsCount()
                     >= connection.getTickets())
                    .count()
            : result.getFreeSeatsCount();
        if (routesSatisfiable > 0
        &&
        result.getFreeSeatsCount() >= connection.getTickets()) {
            Map<String, String> map = new HashMap<>();
            map.put("url", connection.getUrl());
            map.put("routeId", connection.getRouteId());
            notifyUser(connection, map,
            TranslationService
            .getFoundTicketsTranslation(
            connection.getLanguage()));
        }
    } catch (ClientException exception) {
        if(exception.getMessage()
        .contains("Sedadla již nejsou k dispozici")
         ||
         exception.getMessage()
         .contains("Not enough free seats available")
        || exception.getMessage().contains("message:50")) {
            continue;
        }
        log.info(exception.getMessage());
        connection.setType(connection.getType()
        .replace("tickets", ""));
        if (connection.getType().isEmpty()) {
            log.info("Deleting with id " + connection.getId());
            watchDao.deleteById(connection.getId());
        } else {
            log.info("Not watching " + connection.getId());
            watchDao.saveModel(connection);
        }
    }
```

```
            catch (FirebaseMessagingException e) {
                log.info("Notification unsuccessful");
                log.info(e.getMessage());
            }
        }
    }
    @Override
    protected List<WatchedRegioConnection> getWatchedConnections() {
        return
        ((WatchedRegioConnectionDao) watchDao).getTicketConnections();
    }
}
```

We will explain what is happening above. Firstly, we extend the abstract RegioWatchdog and create a correct constructor. We should note here that the parameters in this constructor are automatically provided(wired) by the Spring framework.

Next, we override the watch method, where we iterate through the connections that are loaded via the getWatchedConnections method. In this method, we get all the connections saved in our database that have the type of tickets. For every connection, we first check whether we already did send a notification, if yes, we skip it. The next check is a simple one, we check whether the connection did not arrive yet. If it did, we know we can delete this connection as it does not make sense to watch it anymore. Then, using the RegioJet service mentioned in the previous chapter, we get details about the watched connection, and in them, we find out whether there are enough available seats. This check is done either for the specific seat class if it was requested by the user or for free seats in general. If there are enough free seats, we notify the user. This notification is sent via FirebaseMessaging package, and the whole message is localized based on the saved language of the connection.

In case there are exceptions that are signaling that the route is no longer available to be watched, we delete it and continue to another connection.

```
@Service("FlixDelaysWatchdog")
@Slf4j
public class DelaysWatchdog extends FlixWatchdog {
    public DelaysWatchdog(WatchedFlixConnectionDao watchDao,
     NotificationSenderService sender, FlixService service) {
        super(watchDao, sender, "delays", service);
    }

    @Override
    protected void watch(List<WatchedFlixConnection> connections) {
        for (WatchedFlixConnection connection : connections) {
            try {
                if (wasNotified(connection)) {
                    continue;
                }

                 if (connection.getArrivalTime().after(new Date())) {
                    throw new ClientException(
                    "Connection is already finished");
                }
                StationTimetableDto result = service.getStationTimetable(
                    connection.getStationId(),
                    connection.getArrivalTime().toString(),
                    connection.getArrivalTime().toString()
                );


                Optional<RouteDetailsDto> routesSatisfiable =
                result.getTimetable()
                .getArrivals().stream()
                    .filter(trip -> trip .getTripUid()
                    .equals(connection.getRouteId())
                        &&
                        trip.getDelay() != null)
                    .findFirst();
                if(routesSatisfiable.isEmpty()) {
                    continue;
                }
                if(!routesSatisfiable.get().isHasTracker()) {
                    throw new ClientException("Not tracked");
                }
                Map<String, String> map = new HashMap<>();
                map.put("routeId", connection.getRouteId());
                map.put("delay", routesSatisfiable
                .get().getDelay().getTz());
                notifyUser(connection, map,
                 TranslationService.getFoundDelayTranslation(
                 connection.getLanguage(),
                 routesSatisfiable.get().getDelay().getTz()));
            } catch (ClientException exception) {
                log.info(exception.getMessage());

                //find by ID and delete
                connection.setType(connection.getType()
                 .replace(type, ""));
                if (connection.getType().isEmpty()) {
                    watchDao.deleteById(connection.getId());
                } else {
                    watchDao.saveModel(connection);
                }
            } catch (FirebaseMessagingException e) {
                log.info("Notification unsuccessful");
                log.info(e.getMessage());
            }
        }
    }

    @Override
    protected List<WatchedFlixConnection> getWatchedConnections() {
       return ((WatchedFlixConnectionDao)watchDao).getDelayConnections();
    }
}
```

This next example shows a delay watcher of FlixBus. Firstly, we extend the FlixWatchdog which has an instance of FlixService. The beginning is the same as before, we check whether the notification was not already sent and whether the connection is not already finished. Next, we get details about the connection, but now using the FlixService. For this, we need to check the timetable of the arrival station and filter out all connections that do not have our saved id. If we find the right connection and there is a delay present, we notify the correct user, same as for the RegioJet's Tickets watcher.

5.5 Conclusion

This chapter tries to give a brief overview of the structure and the workings of watchers that are responsible for periodic checks of users' saved connections. Spring functionality that enables us to run scheduled tasks is explained. Lastly, we provided examples and explanations of the actual code of our watchers.

Chapter 6
Future work and deployment

In this chapter, we will talk about the deployment of the web on a server, particularly using cloud services. The next part will be spent discussing the future work that is planned around the back-end service.

6.1 Deployment

To have our app running constantly, we need to deploy it on either our own server or use cloud providers, such as Amazon and Google. As we do not have a dedicated machine to serve as our own server, we went with the deployment into the cloud. We have chosen Amazon Web Services(AWS) as our provider of cloud services, mostly due to the fact that we are familiar with the technology.

„Amazon Web Services (AWS) is a secure cloud services platform, offering compute power, database storage, content delivery and other functionality to help businesses scale and grow.“[17]

AWS provides many different options when it comes to serving your application on the cloud:

- EC2 (Elastic Compute Cloud) — virtual machines in the cloud with OS-level control
- LightSail —„Amazon Lightsail is a virtual private server (VPS) provider ... Lightsail provides developers compute, storage, and networking capacity and capabilities to deploy and manage websites and web applications in the cloud. Lightsail includes everything you need to launch your project quickly – virtual machines, containers, databases, CDN, load balancers, DNS management etc..“[18]
- ECS (Elastic Container Service) — highly scalable container management service that enables you to run Docker containers in the cloud
- EKS (Elastic Container Service for Kubernetes) — highly scalable container management service that enables you to run Kubernetes applications in the cloud
- Lambda — AWS's serverless technology that allows you to run functions in the cloud. It's a huge cost saver as you pay only when your functions execute.
- Batch —batch management that allows you to run hundreds of thousands of computing jobs using other AWS services
- Elastic Beanstalk — „AWS Elastic Beanstalk is an easy-to-use service for deploying and scaling web applications and services developed with Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker on familiar servers such as Apache, Nginx, Passenger, and IIS.“[19]

There are multiple options that would serve our purpose. We could choose EC2 and have a private machine that we can manage by ourselves. However, this would require more experience to configure everything correctly, and would be much more time-consuming. There is another reason. Spring java applications are easily dockerized, meaning we could create a container that our server would be running inside. „A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.“[20].

By dockerizing our application, we can effortlessly control the environment and, most importantly, scale our application by easily creating new instances.

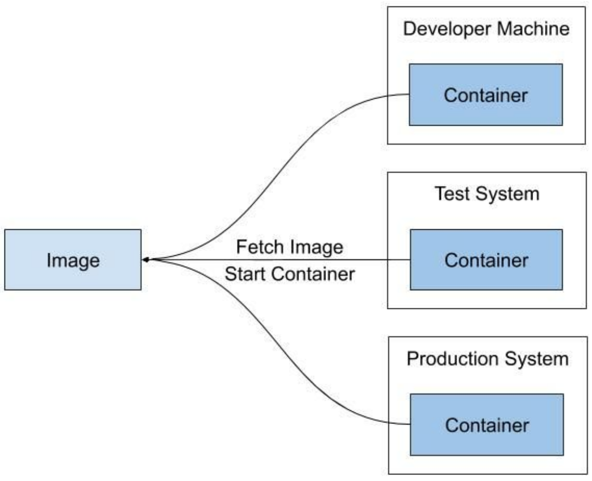


Figure 6.1. Deployment docker[21]

There are three AWS services that make use of docker images: ECS, EKS, and Elastic Beanstalk. We do not want or need to deal with Kubernetes, so only two options remain. Between ECS and Elastic Beanstalk, there are a few differences. The biggest one is that the ECS is more complicated, and one would need to micro-manage it. That is why we decided to go with Elastic Beanstalk. This service is using ECS in itself and provides a simpler interface to it. It does not provide as much control, but it suffices for our purposes.

Firstly, we need to create a docker image of our application. This is easily done by creating a Dockerfile script at the root of our application.

```
FROM openjdk:17-slim-bullseye
RUN adduser --system --group spring
USER spring:spring
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/app.jar"]
```

We faced some problems at this point, due to the fact that the development was done on one of the new Macs that have an M1 processor, which some images still do not support. That is the reason why we selected the base image openjdk:17-slim-bullseye instead of the usual alpine version.

The next step is to build the image and push it to a public repository. We chose the repository available for us in the AWS, called Elastic Container Registry, or ECR.

Once pushed, we now have an image ready to be downloaded and used anywhere we want. Therefore, we are ready to deploy our application on AWS Elastic Beanstalk. To do so we need to utilize docker-compose. „Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration."[22]

We start by creating a docker-compose.yml file, where we define the docker application that our service should use. In our case, it is the image of our Spring application alongside the MongoDB instance.

```
version: "3.9"
services:
  catchit:
    image: 869874504804.dkr.ecr.eu-central-1.amazonaws.com/catchit
    restart: always
    ports:
      - "80:8080"
    depends_on:
      - mongo
  mongo:
    image: mongo:5.0.6
    restart: always
    ports:
      - "27017-27019:27017-27019"
    container_name: mongo
```

After these steps, we are ready to utilize the Elastic Beanstalk command line interface (EB CLI)[23]. Executing commands to create an environment and deploy docker-compose on it is all that is needed to finish the deployment of our application. From the default settings, we have an automatic load balancer that would spin up new instances if the load was too big. However, due to the combined cost of the load balancer and multiple running instances, we have disabled the load balancer and switched to single instance deployment. This way our service will always be available exactly on one node. This is important with the way our database is set up - if we used multiple nodes, the same data would not be available on all of them. More on this in the next part of this chapter.

6.2 Future features

The main improvement to this service would be the addition of new watchers and new providers. Our whole application and code structure have been created with this option in mind, as mentioned in 4.2.1. Therefore, the plans for the future consist of multiple supported providers, as opposed to the current state where only RegioJet and FlixBus are supported.

Apart from adding new providers, there is also a small room for improvement inside the currently implemented providers. We now have an option of watchers for tickets, delays, and for FlixBus the option to watch for cancellation is implemented. However, there is a potential to create more watchers, ones that would be a bit more specific. For this, a detailed analysis of the responses would be needed. For instance, if the required data is there, we could notify users when the platform for the watched connection is selected. Another example watcher could be to watch for changes in either arrival or departure times.

There is one more problem that we mentioned earlier. If we wanted to scale our application by creating new nodes, we would face a problem with our database, where every node would have its own database with different data. We solved this problem for now by configuring AWS to be in single instance mode. However, in the future, the application will hopefully have lots of users, and load balancing would be needed. Therefore, one of the most significant improvements that are planned for the future is to change our database model from the local dockerized database to online services such as MongoDB Atlas[1] or Amazon DocumentDB[2].

6.3 Conclusion

This chapter aims to explain the deployment of our back-end service and the work that is planned for the future. Dockerization of the application is introduced and illustrated with infrastructure examples. A detailed explanation of the AWS cloud services was provided, alongside the reason why we chose the Elastic Beanstalk for our deployment to the cloud. Possible improvements to our implementation were mentioned, with a focus on future changes to the database that are needed for better scalability. This marks the end of the first part of this thesis, which was focused on the back-end side.

[1] `https://www.mongodb.com/atlas/database`
[2] `https://aws.amazon.com/documentdb/`

Chapter 7
Introduction to the mobile application

In the next part of this thesis, we will talk about the mobile application we created as a user interface for our system to communicate with our API. This application enables users to search for RegioJet's connections and buy the tickets or, in case of a sold-out connection, watch for the availability of the tickets. We will begin by explaining the technology we chose alongside the reasons for it. We will mention the problem of having multiple codebases for multiple platforms and how this is solved by tools that enable cross-platform development. We will proceed to illustrate how we designed the application with its screens, theme, and resemblance to the RegioJet's web page in order to be as intuitive to use as we can. Afterward, we will document the results of our user testing. Lastly, we will talk about the deployment of the application on Google Play, why we did not deploy the application on the App Store(yet), and the future work that is planned as the next step for the improvement of the application. In the end, the discussion of our plans to attract users and promote our application will take place.

Chapter 8
Technology

Mobile app development is an area in which we had no experience. That is why, firstly, we wanted to research this field and decide on which technology should we rely. As the mobile market is „one of the most lucrative business venues“[24], there is an abundance of frameworks to choose from. Therefore, in this chapter, we will discuss what technology we chose and why.

8.1 Cross-platform development

With multiple platforms on which the application should run (e.g. Android, iOS), there comes a need to have multiple codebases for the same app. One codebase for Android, one for Apple, et cetera. Now imagine that you wanted to change one text in your application. You would have to change it in all codebases. This is only a small change, but what if the application went through a bigger rework?

That is why cross-platform development is now the prominent solution. With frameworks such as Xamarin, React Native, or Flutter, you write your code once and run it anywhere. Now, it is possible to have one codebase and from it deploy the application to multiple platforms. Of course, native development will still offer more possibilities, but for a simple application, such as ours, this is not a problem. Positives outweigh the negatives, as with native development, we would have to learn multiple languages and frameworks, and write the app in all of them. With cross-platform development, however, we only need to learn and work with one language and one framework.

8.2 Frameworks

After lengthy consideration, we decided to go with a Flutter framework.[1] „Flutter is Google's portable UI toolkit for crafting beautiful, natively compiled applications for mobile, web, and desktop from a single codebase. Flutter works with existing code, is used by developers and organizations around the world, and is free and open source. “[25]

Flutter work with Google's language Dart[2]. „Dart is a client-optimized language for developing fast apps on any platform. Its goal is to offer the most productive programming language for multi-platform development, paired with a flexible execution runtime platform for app frameworks...Dart is designed for a technical envelope that is particularly suited to client development, prioritizing both development (sub-second stateful hot reload) and high-quality production experiences across a wide variety of compilation targets (web, mobile, and desktop).“[26]

With Flutter, we have a single codebase for a range of platforms. It works thanks to Flutter's own high-performance rendering engine. This has another advantage. As

[1] https://flutter.dev/
[2] https://dart.dev/

Flutter is working not with native widgets, but with its own, we can minimize the differences between the platforms, yet still have a native feel for all of them.

We have chosen the Flutter framework for a few reasons:

- Modern - both the framework and the Dart language are modern and open source
- Popular - there is a big community built around the Flutter, see trend on StackOverflow at 8.1
- Well documented - documentation is both clear and exhaustive
- Beginner-friendly - alongside the documentation, there exist multiple tutorials and guides, both fan-made and official. For instance, there is an official example shown at a Keynote talk, that we have found very helpful, and it contained some beautiful animations and assets that we are using in my application[27]
- Feature heavy - A lot of requested features are being continuously released. Also, a lot of functionality and widgets come right out-of-the-box

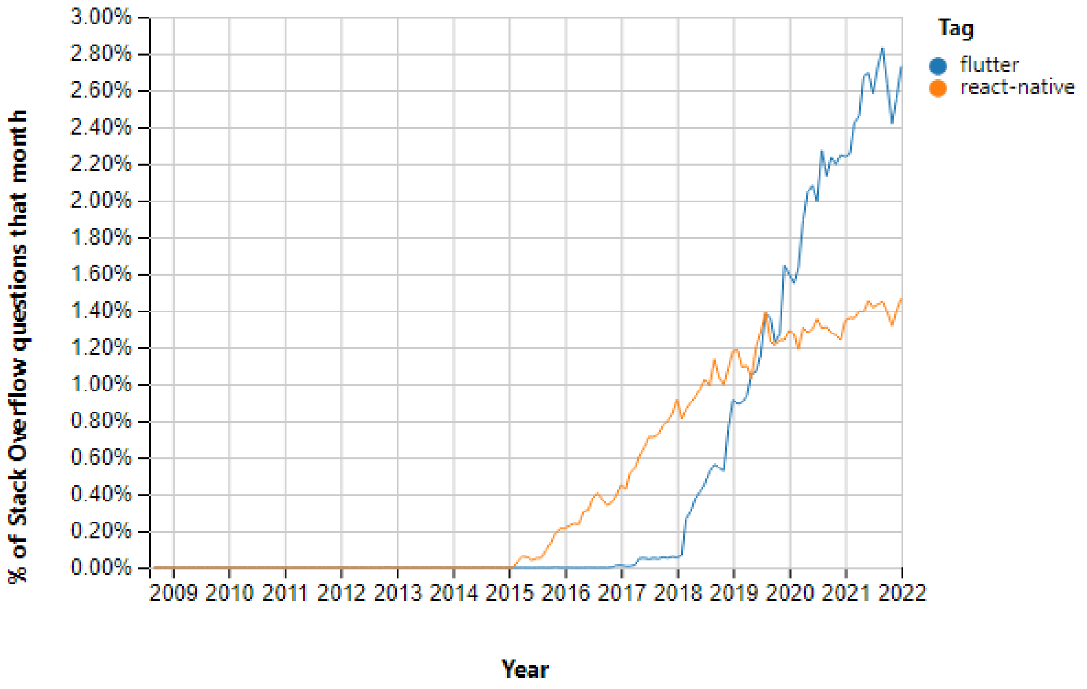


Figure 8.1. Trend on stackoverflow[28]

8.3 Conclusion

In this chapter, we aimed to outline the technology behind mobile app development and lay out the problem with having multiple codebases. We explained the solution in form of cross-platform development. Then we proceed to explain why we chose Flutter as our main technology framework.

Chapter 9
Application design

In this chapter, we will show you the design of our application and the ideas behind it. We will argue that the best way to appeal to users is to strike for a design similar to the one they are used to. In our case, this means creating a user interface(UI) that closely matches the one used on RegioJet's web page. With this in mind, we aimed for recognition rather than recall with our UI. „The big difference between recognition and recall is the amount of cues that can help the memory retrieval; recall involves fewer cues than recognition... Recognition is easier than recall because it involves more cues: all those cues spread activation to related information in memory, raise the answer's activation, and make you more likely to pick it.“[29]

First, we will talk about the individual pages that our application consists of, after which we proceed to compare them to their respective representations on RegioJet's web page.

9.1 Pages

From the get-go, we knew our application should consist of at least four (plus one) separate pages. They are are as follows:

- List of searched routes + details of search parameters
- List of possible seat classes of one route
- List of possible watchers for one seat class
- List of settings

The „plus one“ represents the search page for selecting the departure and arrival stations.

Two additional pages were added during the development. The first was a splash screen, that was added to buy a bit of time for the application to load, and to check whether the device has a working internet connection. The second addition came up during user testing (more on that in 10). Testers mentioned that a page with the list of all watched routes was missing.

We will now go through the pages one by one, and explain the functionality present in them.

9.1.1 Splash screen

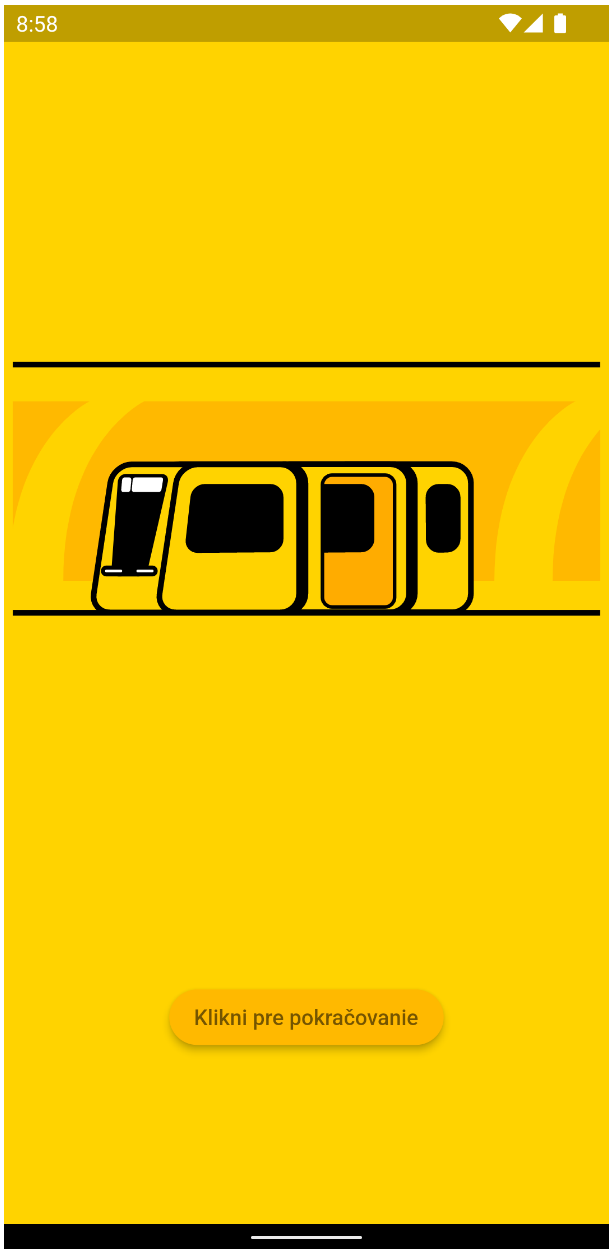


Figure 9.1. Splash screen page

The splash screen serves as the entry point to the application. It shows a random transport vehicle with animations, while the application state is being loaded. There is one more functionality hidden behind the splash screen. Before allowing the user to advance further, it checks whether the device has a connection to the internet and whether the connection works. If not, a message about it is shown to the user. Otherwise, it normally continues into the application.

9.1.2 Details and routes lists

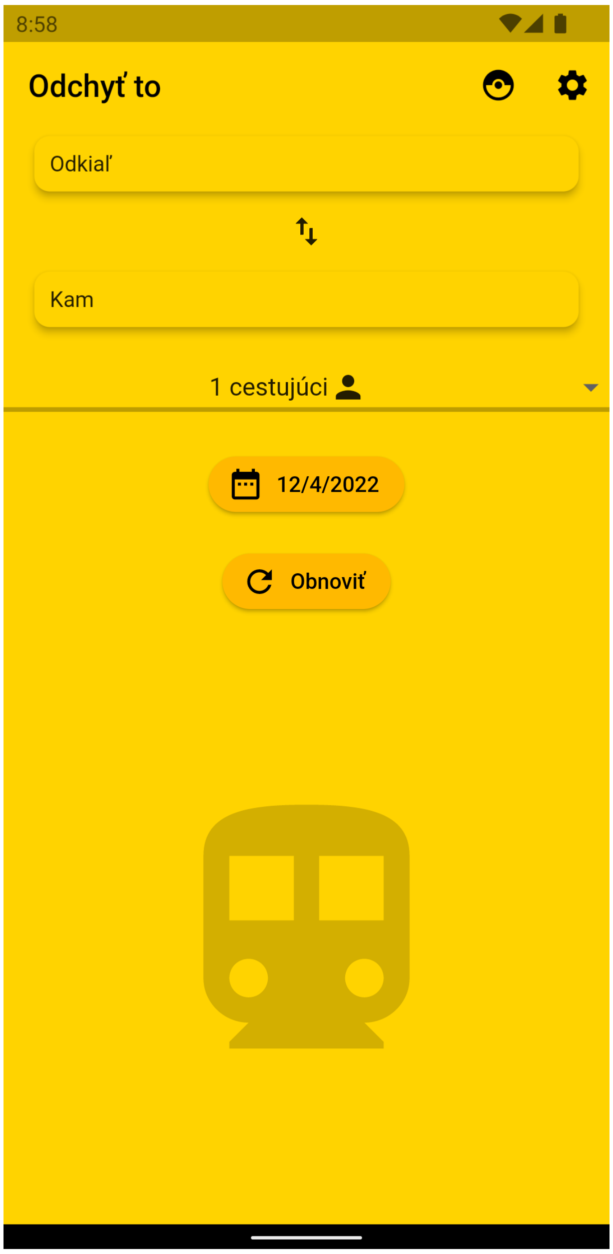
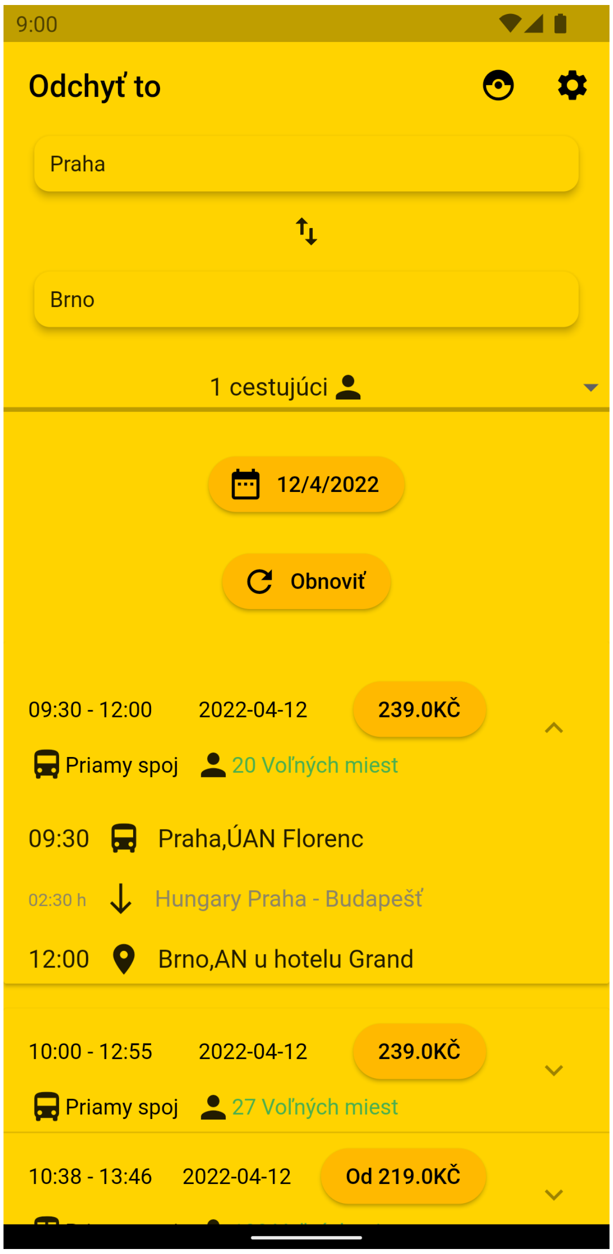


Figure 9.2. Parameter details without routes

Firstly, we have a page with details of search not filled in. This is the first thing user sees after advancing through the splash screen. The top half of the screen is dedicated to the parameters of the route search. There are two fields, each for a departure and arrival station, respectively. Between them is a button to swap these two stations. Under them is a widget for selecting tariffs of users, see 9.11. Then there are the last two buttons. The first one of them is a button to select a date of the route. Upon clicking this button, a calendar widget is shown. The second one serves as a reload button, in case anything goes wrong or the user wants to load the newest data after some time of using the app.



Praha

Brno

1 cestujúci

12/4/2022

09:30 - 12:00 2022-04-12 239.0KČ
Priamy spoj 20 Voľných miest

09:30 Praha,ÚAN Florenc
02:30 h Hungary Praha - Budapešť
12:00 Brno,AN u hotelu Grand

10:00 - 12:55 2022-04-12 239.0KČ
Priamy spoj 27 Voľných miest

10:38 - 13:46 2022-04-12 Od 219.0KČ

Figure 9.3. Parameter details with routes

When all the parameters of the search are filled, an automatic request to get the routes is executed. After the data from our API is fetched, the list of routes is shown in the bottom half of the screen. Each route is shown as an expandable panel. The header contains information about the date and the time when the connection will take place, alongside the information about the transport type(bus vs train vs mixed) and the number of transfers. Next, it contains the number of available seats, and in case of delay, the length of the delay. Lastly, it contains a button with the minimum prize, that upon clicking will take the user onto the next page with a selection of seat classes.

Upon clicking the header, more details are shown. These details include the travel time of each section of the route, with information about the transit type and vehicle and the name of the start and end stations of the section.

9.1.3 Search

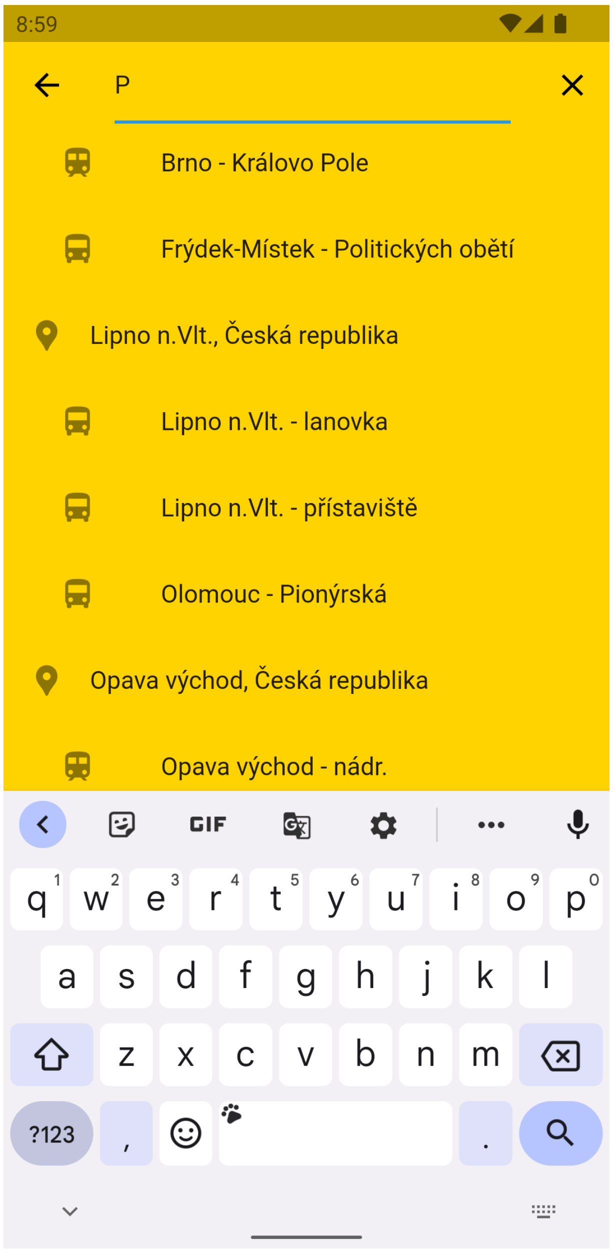
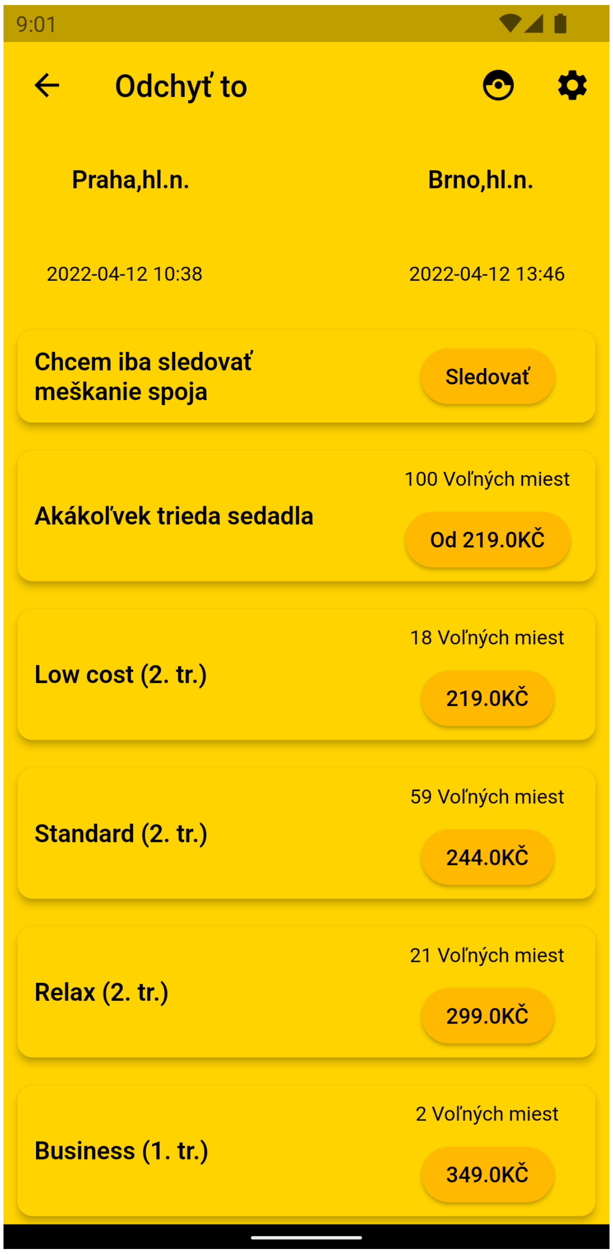

8:59
P
Brno - Královo Pole
Frýdek-Místek - Politických obětí
Lipno n.Vlt., Česká republika
Lipno n.Vlt. - lanovka
Lipno n.Vlt. - přístaviště
Olomouc - Pionýrská
Opava východ, Česká republika
Opava východ - nádr.
GIF
?123

Figure 9.4. Search Page

Next, we have the intermediate search page. This page is shown when a user clicks on either one of the departure/arrival station fields. After typing a character(s), suggestions are shown. Suggestions are sorted alphabetically, with stations from Czech and Slovakia being placed higher. There are two different types of locations. The first is a city type - if selected, it will include all the stations in that city. The second type is one concrete station. Clicking on the suggestion selects it and returns to the previous details page.

9.1.4 Seat classes



9:01

Odchyť to

Praha,hl.n. Brno,hl.n.

2022-04-12 10:38 2022-04-12 13:46

Chcem iba sledovať meškanie spoja — Sledovať

Akákoľvek trieda sedadla — 100 Voľných miest — Od 219.0KČ

Low cost (2. tr.) — 18 Voľných miest — 219.0KČ

Standard (2. tr.) — 59 Voľných miest — 244.0KČ

Relax (2. tr.) — 21 Voľných miest — 299.0KČ

Business (1. tr.) — 2 Voľných miest — 349.0KČ

Figure 9.5. Seat classes page

After selecting a route, seat classes selection is shown. At the top of the screens are details about the route - names of the stations and times of departure and arrival. Directly under them starts a list of seat classes. The first two items are special - they represent no seat class or any seat class, respectively. In case the user selects no seat class, it will automatically start watching for delays on this route.

For the rest of the items, they represent the possible seat classes that RegioJet offers. In case there are enough free seats available in the individual classes, a button with a price is shown. After clicking on it, the user is redirected by the web browser to RegioJet's web where he can reserve and buy the chosen tickets immediately. If the user has an official RegioJet app installed on his device, clicking on the button can take him to the aforementioned app, from where he can buy the tickets.

On the other hand, if there are not enough free seats available, a button to watch this class is shown. Upon clicking this button, the Watchers page is shown.

9.1.5 Watchers

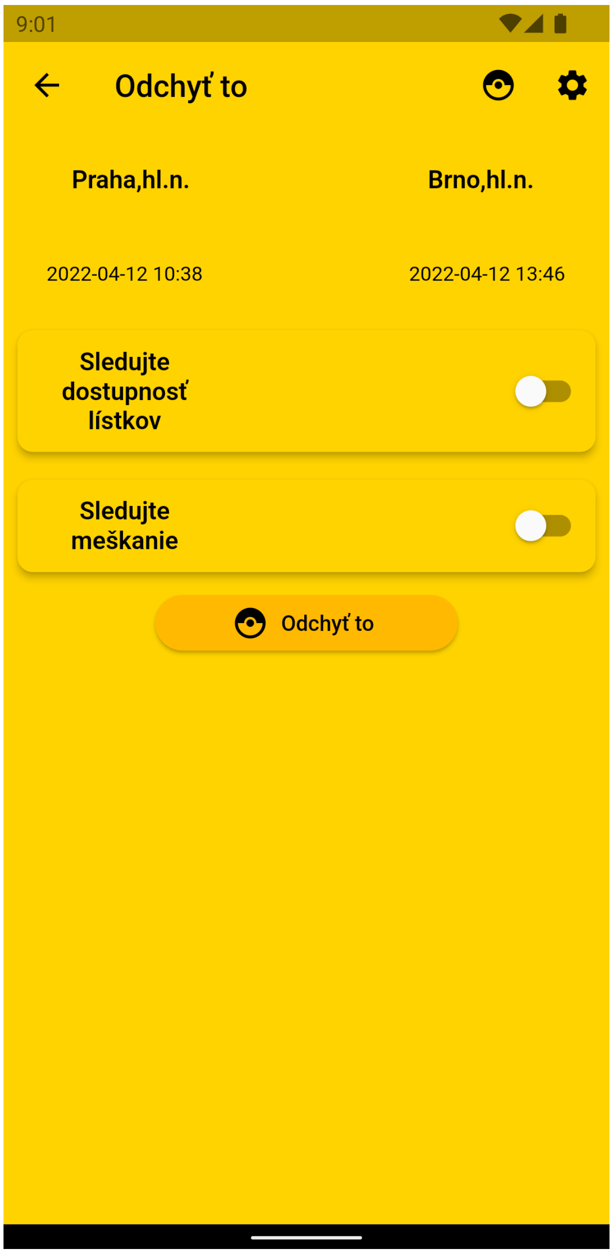
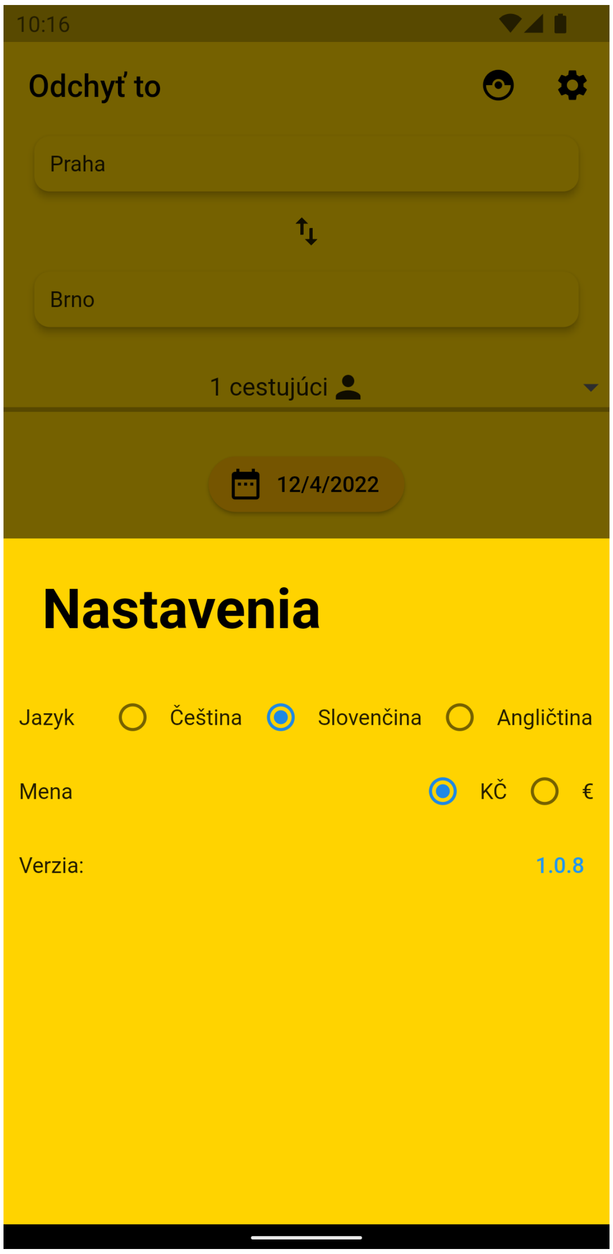


Figure 9.6. Watchers page

The watchers page is the last page in the workflow of creating a watched route. On this page, the possible watchers are displayed to be selected. Right now, there are two watchers implemented, for tickets and for delays. The user can select one or more of them that he wants to apply to the selected route and seat class. After clicking on the select button (with an icon of Pokeball, more on this later), the selected watcher(s) are created.

9.1.6 Settings



Nastavenia

Jazyk ( ) Čeština [✓ Slovenčina] ( ) Angličtina

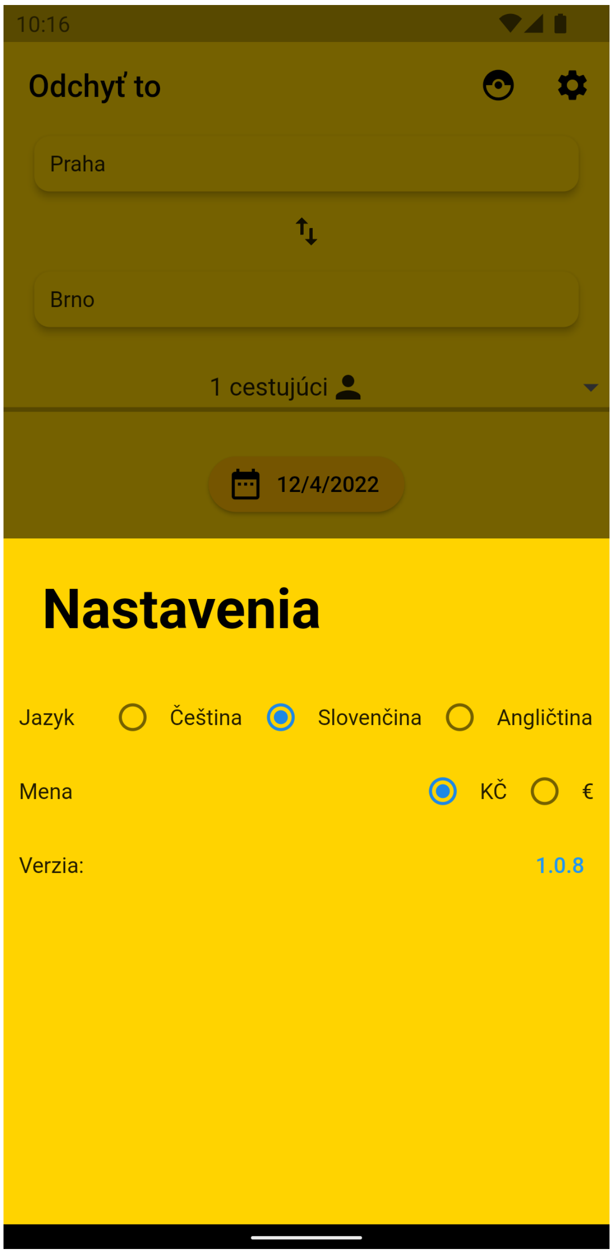
Mena [✓ KČ] ( ) €

Verzia: 1.0.8

Figure 9.7. Settings page

The settings page can be accessed by clicking on the gear icon in the top bar. Within the page, there are language and currency settings. Lastly, we can find the application's version here.

9.1.7 Watched routes



Nastavenia

Jazyk: Čeština [✓ Slovenčina] Angličtina

Mena: [✓ KČ] €

Verzia: 1.0.8

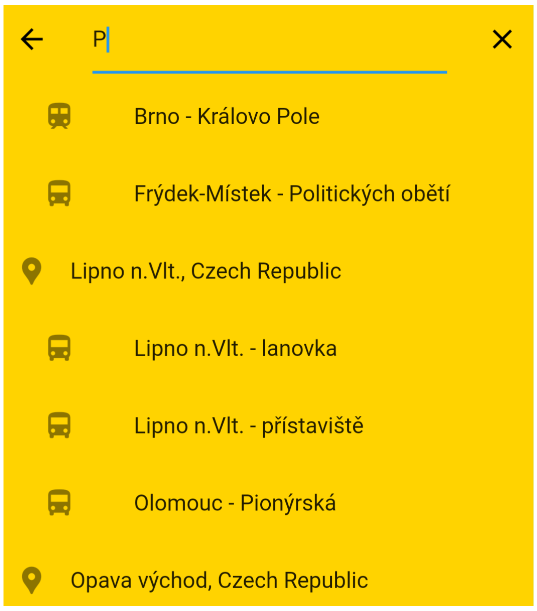
Figure 9.8. Watched routes page

The watched routes page can be accessed by clicking on the Pokeball icon in the top bar. This is an easter egg to the fact that this app's name is Catch It, and the motto of Pokemon is „Catch them all". Within the page, there are routes that we have watched. Due to the difference in data that is available for routes that already left the departure station, there are two lists. The first list is a list of ongoing routes and the second list contains the routes that are planned for the future.

9.2 Desing vs RegioJet's web

As our application is aimed at the users of RegioJet, we tried to stay as close to the design and patterns of RegioJet's web page. As mentioned in the introduction, we want to achieve recollection instead of remembrance for users using our application. This means, that even first-time users should have enough clues from the original web page to recall how to use it. To complement this, the theme of the application was set to be yellow, similar to the RegioJet, and the other colors were selected to be matching this theme.

Other than that, we also tried to use the same patterns of icons, widgets, and texts. Let us show you some comparisons side to side:


P

Brno - Královo Pole

Frýdek-Místek - Politických obětí

Lipno n.Vlt., Czech Republic

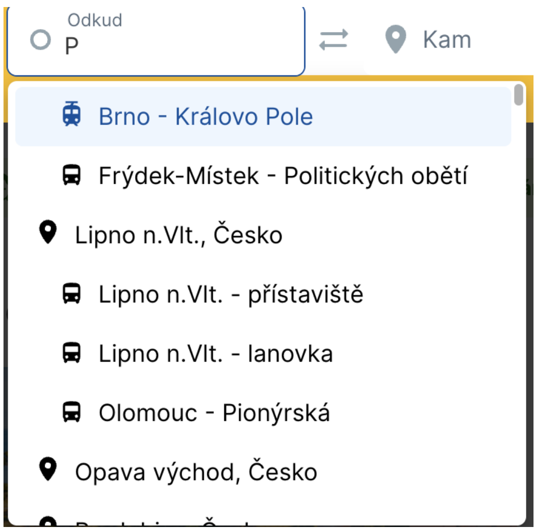
Lipno n.Vlt. - lanovka

Lipno n.Vlt. - přístaviště

Olomouc - Pionýrská

Opava východ, Czech Republic

Figure 9.9. App's locations search


Odkud
P

Kam

Brno - Královo Pole

Frýdek-Místek - Politických obětí

Lipno n.Vlt., Česko

Lipno n.Vlt. - přístaviště

Lipno n.Vlt. - lanovka

Olomouc - Pionýrská

Opava východ, Česko

Figure 9.10. RegioJet's locations search

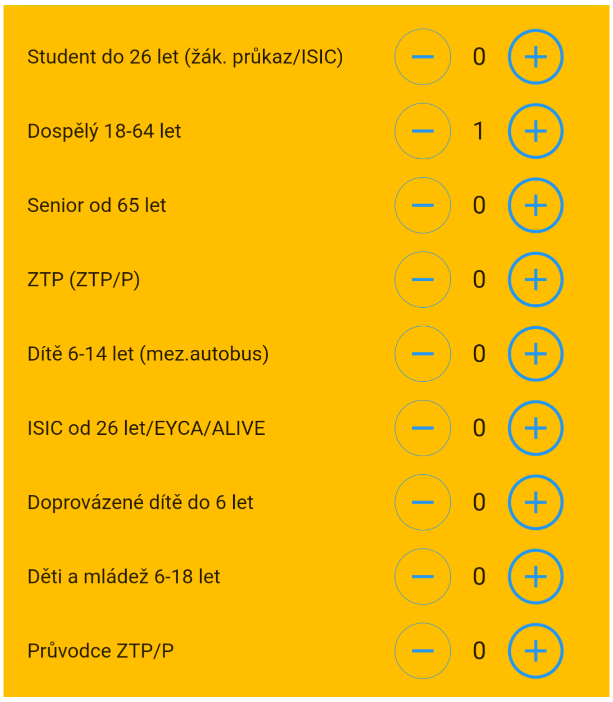
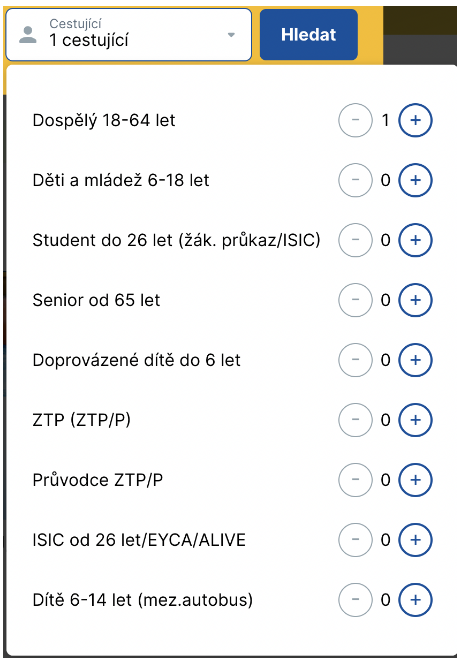

Student do 26 let (žák. průkaz/ISIC)	0
Dospělý 18-64 let	1
Senior od 65 let	0
ZTP (ZTP/P)	0
Dítě 6-14 let (mez.autobus)	0
ISIC od 26 let/EYCA/ALIVE	0
Doprovázené dítě do 6 let	0
Děti a mládež 6-18 let	0
Průvodce ZTP/P	0

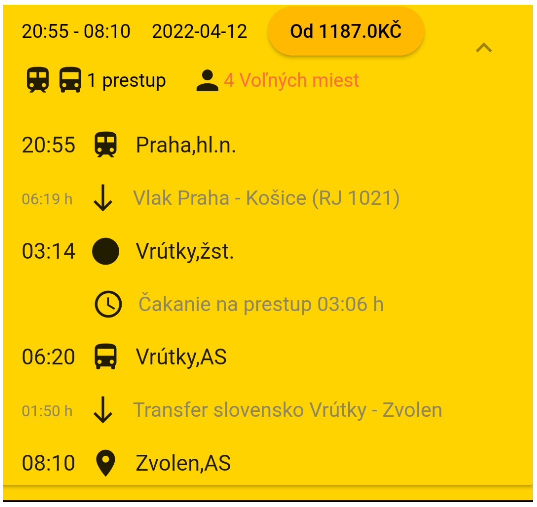
Figure 9.11. App's tariffs


Cestující
1 cestující

Hledat

Dospělý 18-64 let	1
Děti a mládež 6-18 let	0
Student do 26 let (žák. průkaz/ISIC)	0
Senior od 65 let	0
Doprovázené dítě do 6 let	0
ZTP (ZTP/P)	0
Průvodce ZTP/P	0
ISIC od 26 let/EYCA/ALIVE	0
Dítě 6-14 let (mez.autobus)	0

Figure 9.12. RegioJet's tariffs



20:55 - 08:10 2022-04-12 Od 1187.0KČ

1 prestup 4 Voľných miest

20:55 Praha,hl.n.

06:19 h Vlak Praha - Košice (RJ 1021)

03:14 Vrútky,žst.

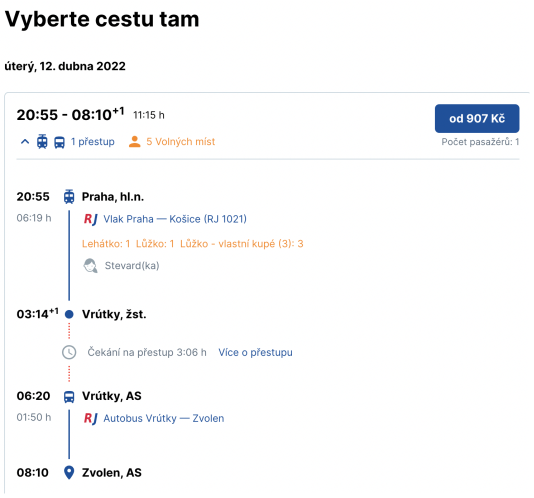
Čakanie na prestup 03:06 h

06:20 Vrútky,AS

01:50 h Transfer slovensko Vrútky - Zvolen

08:10 Zvolen,AS

Figure 9.13. App's connection details



Vyberte cestu tam

úterý, 12. dubna 2022

20:55 - 08:10^{+1} 11:15 h

od 907 Kč

1 přestup 5 Volných míst

Počet pasažérů: 1

20:55 Praha, hl.n.

06:19 h RJ Vlak Praha — Košice (RJ 1021)

Lehátko: 1 Lůžko: 1 Lůžko - vlastní kupé (3): 3

Stevard(ka)

03:14^{+1} Vrútky, žst.

Čekání na přestup 3:06 h Více o přestupu

06:20 Vrútky, AS

01:50 h RJ Autobus Vrútky — Zvolen

08:10 Zvolen, AS

Figure 9.14. RegioJet's connection details

9.3 Localization

As RegioJet is active in multiple countries, so should be our application. For this reason, the app is fully localized, so far, in three languages: Slovak, Czech, and English. The initial language is taken from the device. If this language is not supported, the Czech language is taken as a default one. However, if the user changes the language in the settings, this language is saved and used in the subsequent usages of the application.

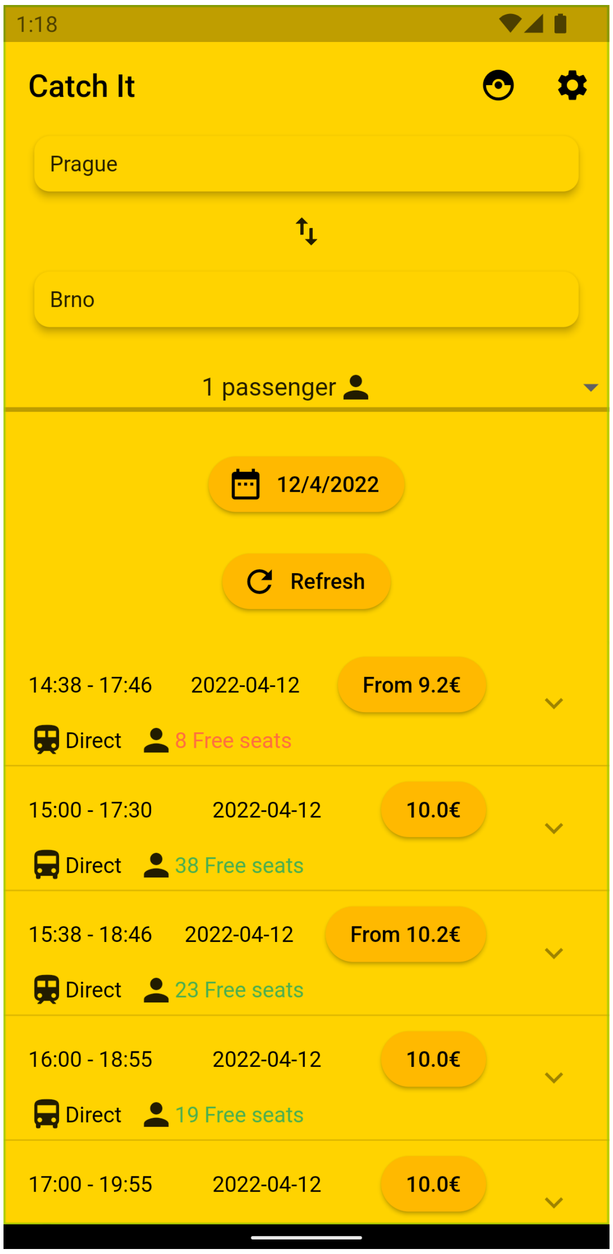
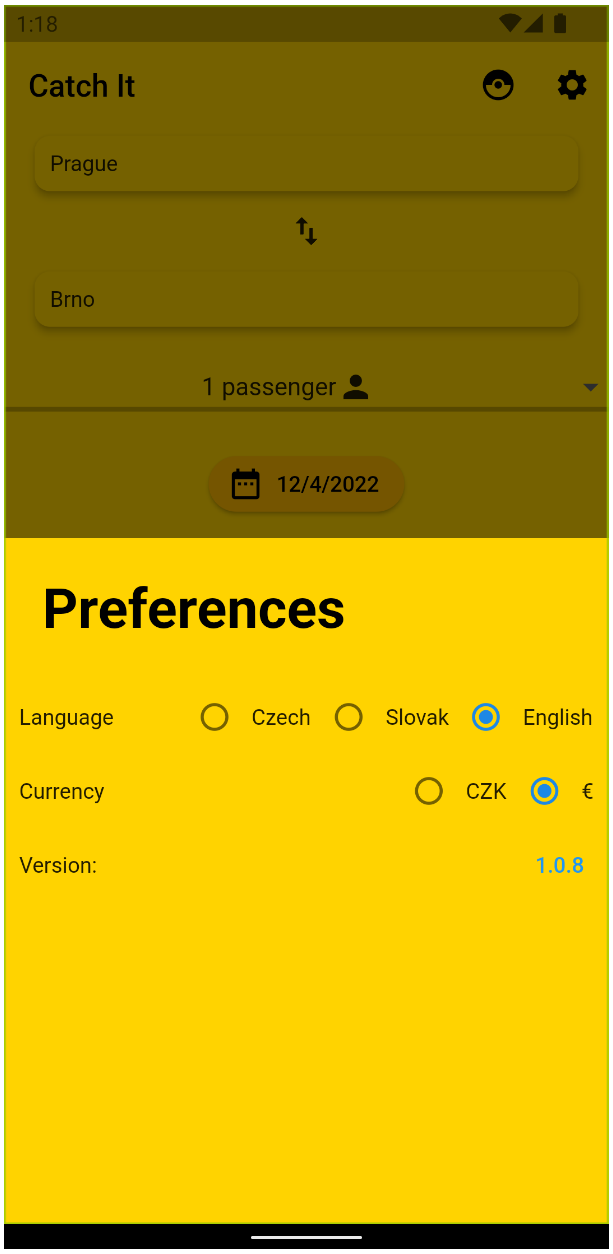


Figure 9.15. Routes in English

As we are not only dealing with text, but with price as well, to complete the localization, the application supports multiple currencies. At the moment, the Czech crown and Euro are supported. Crown is selected by default, but if the user changes the currency, it behaves the same as the language mentioned above.



Preferences

Language ○ Czech ○ Slovak [✓ English]

Currency ○ CZK [✓ €]

Version: 1.0.8

Figure 9.16. Settings in English

9.4 Conclusion

In this chapter, we discussed the design of the mobile application. We showed you all the pages that the app consists of, with an explanation of their functions. We compared the design to RegioJet's web page and argued that we aimed for a recollection rather than the remembrance with the user interface of the application. Lastly, we talked about the localization of our app.

Chapter 10
User Testing

In this chapter, we will talk about the user testing that we conducted. This was done to ensure the UI is intuitive and the transition from RegioJet's web page is seamless. Another reason was to get the opinions of other people on whether the design fulfilled the goals outlined in the 9. The tests were taken on the Android version of the application.

10.1 Form

The testing was split up into three parts. In the first one, general questions about users' experience with RegioJet were asked. The second part consisted of five tasks that the users were supposed to complete. The time to finish each task was measured. In the last part, the participants talked about the app, whether they missed any functionality and the testing process.

We will now state the questions and tasks of the testing: The first part:

- Have you used RegioJet services?
- How often do you travel via RegioJet?
- Have you encountered a situation, when the tickets you wanted were sold-out?

The second part:

- Find any route from Prague to Brno(any station) that takes place on May the 2nd (2.5.2022)
- Find any route from Brno to Prague(any station) that takes place on May the 2nd (2.5.2022)
- Change the language of the application
- Find a train route from Prague to Brno(any station) that takes place on May the 2nd (2.5.2022). Create a watcher for delays
- Delete the watched route from the previous step

The last part:

- Do you have any notes about the application? Did you miss any functionality?

In the next section, we will go over the participants and their responses one by one.

10.1.1 Participant 1

The first tester has used RegioJet in the past and uses it sever times a year. He even encountered the situation when he wanted to purchase a ticket, but they were all sold out.

For the timed tasks, it took him thirty seconds to find routes from Prague to Brno, and then twenty-five seconds to find them from Brno to Prague. To change the language of the application, only eight seconds were needed. Then, to create a delay watcher for connection from Prague to Brno, it took thirty-five seconds, and the watcher was deleted in ten seconds.

He liked the application, and when asked about what future he thought could be missing, he explained that a page with the list of all watched routes would be nice to have. He did not have any further complaints.

10.1.2 Participant 2

The second user also used RegioJet several times per year and faced the issue of tickets being sold out, too.

She spent less time on the first task, namely twenty-six seconds. The second task was interesting because at first, it took her about twenty seconds, but then she realized she could have just switched the arrival and departure station using the switch button. We allowed her to try it again, and this time the task was finished in five seconds. Changing the language took her nineteen seconds. A lot more time was spent on setting the watcher for the third task - one minute and two seconds. Then she proceeded to delete this watcher in thirteen seconds.

When asked about the process and the application, she had something to say. We present her response: „The application is pretty intuitive. It took me longer than necessary to switch routes. The first time I did it manually instead of simply using the button. Also, I got the longest time for setting watcher because it took longer to reload all the information. Other tasks were done in matter of seconds, each function was simple and clear to find.“

10.1.3 Participant 3

The third tester uses RegioJet regularly, once a month, yet in his experience, the tickets he wanted to purchase were always available.

He finished the first task in fifteen seconds. For the second task, he realized there is a button providing just the wanted outcome, and he did it in five seconds. This same time it took him to change the language. He was the quickest of them all, setting the watcher in just twenty seconds. To delete the watcher, it took him exactly half of the time it took him to set it up.

He had further comments.

10.1.4 Participant 4

The last participant uses a RegioJet, but only once a year. However, he too experienced that the tickets he wanted to purchase were sold out.

It took him twenty-five seconds to find the routes from Prague to Brno, but only six seconds to find it the other way around. Changing the app's language was even faster, with the result of five seconds. Creation and deletion of the watcher took him longer, thirty-eight and twenty-five seconds, respectively. These higher times were partly due to him encountering longer loading times.

He mentioned this in the last section, where he complained that to delete the watcher he had to delete it from the route list that had to be loaded again.

10.2 Summary

All in all, the users thought the application was intuitive and they knew how to finish the task without us giving them any hints. Most of them realized that the second task could be finished with just one click of a button. Only one did not realize it until we told him at the end of the testing.

Thanks to their feedback, we were able to realize quickly that accessing watchers only from the list of routes is not ideal, and we created a new page with all the watched routes, as mentioned in 9.1.7.

10.3 Conclusion

In this chapter, we discussed the user testing that took place with four participants. Firstly, we explained the form of this testing and stated the questions and tasks asked of all users. Then we proceeded to go over their responses one by one. Lastly, we summarized the findings and the impact this testing had on our application, namely the addition of a page with a list of all watched routes.

Chapter 11
Future work and deployment

In this chapter, we will talk about the deployment of the mobile application on Google Play. The next part will be spent discussing the future work that is planned for the app. Lastly, we will talk about the methods to promote the application to users' attention.

11.1 Publishing

11.1.1 Google Play Store

To publish our application, we followed the official Flutter guideline[1].

Firstly, we had to create a developer's account, and pay a one-time fee of 25 dollars. With this done, the next step was to create a new upload keystore that is required for new app bundles. Then we changed the Gradle configuration to actually use this keystore for signing the app bundle. There is one more change needed. Android applications do not have permission to use the internet automatically. We need to add it by adding the following lines to AndroidManifest.xml

```
<manifest xmlns:android="...">
  <uses-permission android:name="android.permission.INTERNET"/>
</manifest>
```

Now, the application is ready. After running

```
flutter build appbundle
```

we have the .aab bundle stored in build/app/outputs/bundle/release/.

The next steps are easy enough, but a bit tedious. We need to set everything up for a successful release to Google Play. This includes adding an app description, logo, screenshots, translations, and selecting in which countries the app should be available. After all required fields are filled, we can go to Production and create a new release. Here we need to upload the app bundle from the previous step and create the release. Now all that is left is to wait for them to check and review the application. This can take several days. Then, as soon as they approve it, the app will start rolling out to Google Play. This too can take up some time, so there is no reason to panic if you cannot find your application there right away. Not, the application is available for download on Google Play[2].

11.1.2 App Store

As we mentioned in 8.1, the technology was chosen so we can deploy our application to multiple platforms from a single codebase. And as such, we have a working version for both Android and iOS, and actually, the app can run on Apple Macs that have M1 processors.

[1] https://docs.flutter.dev/deployment/android
[2] https://play.google.com/store/apps/details?id=cz.catchit

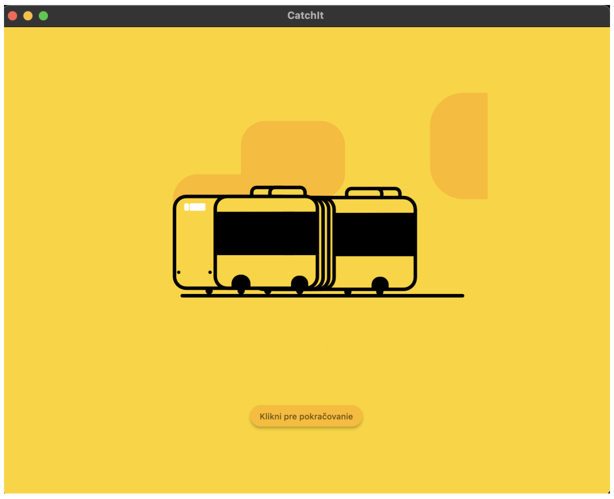


Figure 11.1. App running on Mac

However, the process of publishing the app to the Apple Store requires developers to have a Mac computer and pay a yearly fee of 3000 crowns. Thankfully, as part of the agreement between Czech Technical University and Apple, we as students are eligible to set up our developer account without paying the fee under their organization.

To publish our application onto App Store, we followed the official Flutter guideline for iOS[1]. The requirement is to have a Mac that would be used for signing the application. The process is straightforwards, thanks to automatic signing by Xcode[2]. We just had to create a new application at App Store connect and run

```
flutter build ipa
```

This created a .xcarchive that has to be opened using Xcode. Once opened, there is an option to distribute the app to App Store. In the process of validating the app, there is an option to choose between manual and automatic signing. We chose the latter, as it sufficed for our purposes and it made the publishing much easier.

After the upload is completed, we have to fill out all the required information, similar to the Google Play mentioned above. We needed to provide screenshots of our app on iPhones and iPad of different sizes, add names and descriptions for all supported languages, take a privacy survey and for each language, provide a link to a page with support to your page.

After all of the above is filled in, we submitted our app for review. In our case, the application was approved and published to the App Store in a matter of hours, compared to the few days it took Google to finish the review process. It is now available for download on the App Store[3].

11.2 Future features

There is one big improvement to the application that we have in mind. Supporting not only RegioJet but also FlixBus, as we have our back-end ready for both of them. Therefore, in the future, we want to add a tab for the providers, and users could freely switch between them in one app. However, to correctly create a design for FlixBus

[1] `https://docs.flutter.dev/deployment/ios`
[2] `https://developer.apple.com/xcode/`
[3] `https://apps.apple.com/us/app/odchy%C5%A5-to/id1619323306`

and implement is almost the same (in the amount of work) as creating a whole new application.

For minor additions, we plan on supporting the German language and adding some customization to the app, such as the interval of notifications.

Of course, any improvements to the back-end service, like new watchers, will be also reflected in the application.

11.3 Promotion

After this thesis is finished and presented, we would like to promote our app to a wider range of users. To do so, we can take advantage of the fact that potential users of our application should be traveling often, and even better if they periodically travel between Czechia and Slovakia. Fortunately, there are a lot of Facebook groups that fit our description perfectly. Groups such as Slovaks in Prague[1] and similar, grouping people of Slovak nationality living in the Czech Republic, are perfect for us. They even contain posts about RegioJet's connections or selling tickets for a specific date. We would promote our application with posts in these kinds of groups, and of course, by showing the application to our friends that could use an app like this, hoping that if they like it, they will continue the promotion chain.

11.4 Conclusion

This chapter aims to explain the deployment of our application and the work that is planned for the future. The process of enrolling the app into the Google Play Store and Apple App Store is discussed. Improvements that are planned, such as the addition of support for the German language, the addition of FlixBus provider, or customization of the app were proposed. Lastly, we introduced ways to promote our application to get users' traction.

With this, we conclude the second part of this thesis, which was dedicated to the mobile application consuming the data and API from the first part.

[1] `https://www.facebook.com/groups/slovacivprahe`

Chapter 12
Conclusion

The purpose of this thesis was to propose and implement a solution to the problem of booking sold-out tickets from public transportation providers.

We formulated the current problem of having to periodically visit a web page of the provider and check whether someone canceled their ticket or the provider increased the capacity. We addressed this issue and argued that the best solution is to automate this process in form of a mobile application.

Firstly, we talked about the process of analyzing network communications of the current providers' reservations systems. Then we proceeded to show how to use this analysis to create a back-end service that would emulate their functionality and expose their data for our use in the form of a mobile application. We presented the design, technology, and architecture of this server implementation by using REST API and then followed with the implementation, describing how it aligned with the proposed architecture, and how it fulfilled the laid down goals. We explained the deployment of our server into the cloud by using the AWS Elastic Beanstalk service, alongside the discussion of future improvements that are planned. Lastly, we talked about the development, design, and testing of this cross-platform mobile application, as well as about the technology behind it. We talked about the process of deploying our application into the Google Play Store and also how the process was different for the Apple Store. In the end, we mentioned possible improvements to the app, in particular, the addition of support for FlixBus.

Appendix A
Symbols

API	Application Programming Interface
AWS	Amazon Web Services
DI	Dependency injection
DTO	Data Transfer Object
EB CLI	Elastic Beanstalk command line interface
EC2	Elastic Compute Cloud
ECR	Elastic Container Registry
ECS	Elastic Container Service
EKS	Elastic Container Service for Kubernetes
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IoC	Inversion of Control
JSON	JavaScript Object Notation
NoSQL	Not Only Structured Query Language
POJO	Plain Old Java Object
REST	Representational State Transfer
SQL	Structured Query Language
TTL	Time to live
UI	User Interface
URI	Uniform resource identifier
URL	Uniform resource locator
XML	eXtensible Markup Language
YAML	Yet another markup language

References

[1] *Spring Framework.* https://spring.io/projects/spring-framework. Last accessed on on 2022-01-05.

[2] *Why Spring?* https://spring.io/why-spring. Last accessed on on 2022-01-05.

[3] *Relational vs. non-relational databases, 2020-08-13.* https://www.pluralsight.com/blog/software-development/relational-vs-non-relational-databases. Last accessed on on 2022-04-01.

[4] *What's the Difference? Relational vs Non-Relational Databases, 2021-02-15.* https://insightsoftware.com/blog/whats-the-difference-relational-vs-non-relational-databases/. Last accessed on on 2022-04-01.

[5] *What Is a REST API?, 2020-11-20.* https://www.akana.com/blog/what-is-rest-api. Last accessed on on 2022-04-01.

[6] *REST Architectural Constraints, 2022-03-09.* https://restfulapi.net/rest-architectural-constraints/. Last accessed on on 2022-04-13.

[7] *Clean API Architecture, 2019-06-01.* https://medium.com/perry-street-software-engineering/clean-api-architecture-2b57074084d5. Last accessed on on 2022-04-01.

[8] *Data transfer object, 2021-03-31.* https://en.wikipedia.org/wiki/Data_transfer_object. Last accessed on on 2022-04-01.

[9] *Generics in Java, 2022-02-09.* https://www.geeksforgeeks.org/generics-in-java/. Last accessed on on 2022-04-13.

[10] *What is Caching and How It Works, 2021-05-17.* https://auth0.com/blog/what-is-caching-and-how-it-works/. Last accessed on on 2022-04-05.

[11] *Caching.* https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/boot-features-caching.html. Last accessed on on 2022-02-22.

[12] *Home, 2021-02-22.* https://github.com/ben-manes/caffeine/wiki. Last accessed on on 2022-04-05.

[13] *Separation of Concerns in Software Design, 2020-01-16.* https://nalexn.github.io/separation-of-concerns/. Last accessed on on 2022-04-13.

[14] *Spring Boot Integration with MongoDB Tutorial.* https://www.mongodb.com/compatibility/spring-boot. Last accessed on on 2022-04-06.

[15] *MongoDB repositories.*
https://docs.spring.io/spring-data/mongodb/docs/1.2.0.RELEASE/reference/html/mongo.repositories.html. Last accessed on on 2022-04-06.

[16] *Firebase Cloud Messaging, 2022-03-24.*
https://firebase.google.com/docs/cloud-messaging. Last accessed on on 2022-04-08.

[17] *What is AWS and What can you do with it, 2018-06-03.*
https://medium.com/@kunalyadav/what-is-aws-and-what-can-you-do-with-it-395b585b03c. Last accessed on on 2022-04-07.

[18] *Amazon Lightsail FAQs.*
https://aws.amazon.com/lightsail/faq/. Last accessed on on 2022-04-09.

[19] *AWS Elastic Beanstalk.*
https://aws.amazon.com/elasticbeanstalk/. Last accessed on on 2022-04-09.

[20] *What is a Container?*
https://www.docker.com/resources/what-container.

[21] *How to dockerize your PHP application for AWS Fargate?*
http://cloudonaut.io/how-to-dockerize-your-php-application-for-aws-fargate.

[22] *Overview of Docker Compose.*
https://docs.docker.com/compose/. Last accessed on on 2022-04-09.

[23] *Using the Elastic Beanstalk command line interface (EB CLI).*
https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3.html. Last accessed on on 2022-04-09.

[24] *Mobile Application Development Statistics: 5 Facts, 2021-11-23.*
https://intersog.com/blog/mobile-app-development-statistics/. Last accessed on on 2022-04-13.

[25] *FAQ.*
https://docs.flutter.dev/resources/faq. Last accessed on on 2022-04-11.

[26] *Dart overview, 2022-03-23.*
https://dart.dev/overview. Last accessed on on 2022-04-11.

[27] *GitHub Repository, 2019-06-05.*
https://github.com/mjohnsullivan/berlin_transport/. Last accessed on on 2022-04-13.

[28] *Stack Overflow Trends.*
https://insights.stackoverflow.com/trends?tags=flutter%2Creact-native. Last accessed on on 2022-04-11.

[29] *Memory Recognition and Recall in User Interfaces, 2014-06-06.*
https://www.nngroup.com/articles/recognition-and-recall. Last accessed on on 2022-04-12.

[30] C. Martin, Robert. *Clean Code: A Handbook of Agile Software Craftsmanship.* Edition 1 ed. Prentice Hall, 2008. ISBN 978-0132350884.

[31] Roger S. Pressmann, Bruce Maxim. *Software Engineering: A Practitioner's Approach.* 8th edition ed. McGraw Hill, 2014. ISBN 978-0078022128.

[32] Gaitatzis, Tony. *Learn REST APIs: Your guide to how to find, learn, and connect to the REST APIs that powers the Internet of Things revolution.* 8th edition ed. BackupBrain Press, 2019. ISBN 978-1989775004.

[33] Sharma, Sourabh. *Modern API Development with Spring and Spring Boot: Design highly scalable and maintainable APIs with REST, gRPC, GraphQL, and the reactive paradigm*. Packt Publishing, 2021. ISBN 978-1800562479.

[34] Simone Alessandria, Brian Kayfitz. *Flutter Cookbook: Over 100 proven techniques and solutions for app development with Flutter 2.2 and Dart*. Packt Publishing, 2021. ISBN 978-1838823382.