Assignment of master's thesis

Title:	System for monetization of carsharing data
Student:	Bc. Ondřej Cihlář
Supervisor:	Ing. Václav Jirovský, Ph.D.
Study program:	Informatics
Branch / specialization:	Managerial Informatics
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

The Uniqway system collects various anonymised data on vehicle movements. Today, many other services and systems are based on knowledge of the behaviour of individuals or groups in society. The aim of the thesis is:
1. to enrich vehicle provided data with data acquired from mobile phones of users riding in a road vehicle (use the existing work of transport mode detection as a basis);
2. to create a public interface exposing relevant anonymised floating car/user data;
3. to propose how could be external entities charged for data.
Collaborate with the Uniqway student team to implement the work.

Electronically approved by Ing. David Buchtela, Ph.D. on 6 November 2022 in Prague.

Master’s thesis

SYSTEM FOR MONETIZATION OF CARSHARING DATA

Bc. Ondřej Cihlář

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Václav Jirovský, Ph.D.
May 4, 2023

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Bc. Ondřej Cihlář. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Cihlář Ondřej. *System for monetization of carsharing data*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

Acknowledgments **vi**

Declaration **vii**

Abstract **viii**

Abbreviations **ix**

Introduction **1**

1 Theoretical part **3**
1.1 Application programming interface 3
1.1.1 API endpoint 4
1.1.2 API types 4
1.2 REST API 5
1.2.1 Resource identification 5
1.2.2 HTTP protocol 7
1.3 API security 9
1.3.1 Authentication 9
1.3.2 Access control 9
1.3.3 API keys 10
1.4 The Uniqway system technologies 11
1.4.1 Java Technology 11
1.4.2 Play framework 11
1.4.3 PostgreSQL 12
1.4.4 PostGIS 12
1.4.5 Python 12
1.4.6 Elasticsearch 13

2 Requirements analysis and solution design **17**
2.1 The Uniqway system architecture 17
2.2 The Uniqway data analysis 18
2.2.1 PostgreSQL database data 18
2.2.2 Elasticsearch data 19
2.2.3 User transport mode data 19
2.3 Requirement analysis 20
2.3.1 Functional requirements 20
2.3.2 Non-functional requirements 20
2.4 Use case model 21
2.4.1 List of actors 21
2.4.2 List of use cases 22
2.4.3 Use case diagram 24
2.5 Solution design 25
2.5.1 API users and API keys 25

2.5.2 User transport data 26
2.5.3 Exposure of data for public API 27

3 Implementation **29**
3.1 API users and API keys 30
3.1.1 Persistence layer 30
3.1.2 Generation of API key 30
3.1.3 Services 32
3.1.4 API key authentication 32
3.1.5 Controllers and admin API 33
3.2 User transport data 35
3.2.1 Persistence layer 35
3.2.2 Services 36
3.2.3 Controllers and admin API 36
3.3 Public API 38
3.3.1 Request factory 38
3.3.2 Pagination 39
3.3.3 Filters 40
3.3.4 Search services 42
3.3.5 Controllers and API 44
3.4 Testing 47
3.4.1 Unit test 47
3.4.2 Python API tests 49

4 API monetization **51**
4.1 Free model 51
4.1.1 Freemium model 51
4.2 Fee-based model 52
4.3 Revenue-sharing model 52
4.4 Monetization model for the Uniqway public API 52

5 Conclusion **53**

Obsah přiloženého média **57**

List of Figures

1.1 Communication of client and server through the API[3] 3
1.2 Types of APIs [5] 4
1.3 API key usage [17] 10
1.4 The Java platform [18] 11

2.1 The Uniqway system architecture 18
2.2 Use case diagram 24
2.3 API users model 25
2.4 User transport data model 26
2.5 Data retrieval 27

List of Tables

List of code listings

3.1 API user entity class 30
3.2 RandomHexStringFactory class 31
3.3 generateRandomApiKey method 31
3.4 API key is generated in a synchronized block of code 32
3.5 ApiKeyAuthenticatorAction call method 33
3.6 UserTransportDao findUserTransportsInGeoDistance method 36
3.7 Creation of basic search query 38
3.8 Creation of match query 38
3.9 Creation of range query 39
3.10 Creation of geo query 39
3.11 UserTransport getFilters method 41
3.12 ElasticFilterSortApplyService applyFiltersForSearchRequest method 41
3.13 ElasticFilterSortApplyService applyDateRangeFilterForSearchRequest method . 42
3.14 ElasticSearchRequestFactory buildSearchRequest method 42
3.15 CarRideStopService 43
3.16 ApiKeyGenerateService unit test 48

I want to thank my tutor, Ing. Václav Jirovský, Ph.D., and the Uniqway student team for guiding my work and helping me in the search for a solution. I also want to thank my family and friends for their support during my studies.

Declaration

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

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

In Prague on May 4, 2023 ....

Abstract

This thesis deals with the analysis of data collected by the carsharing startup Uniqway. The anonymized data on vehicle and user movements is exposed within a public API. An API key is used to authenticate the users of Uniqway's public data. For future monetization of the exposed data, models based on the volume of data provided and the number of transactions performed are analyzed and proposed.

Keywords public API, Uniqway, API design, data monetization, carsharing service, API key authentication, REST API, Java, Play framework

Abstrakt

Tato práce se zabývá analýzou dat, které sbírá carsharingový startup Uniqway. Anonymizovaná data o pohybu vozidel a uživatelů jsou vystavena v rámci veřejného API. Pro autentizaci uživatelů veřejných dat Uniqway je použit API klíč. Pro budoucí monetizaci vystavených dat jsou analyzovány a navrženy modely založené na objemu poskytnutých dat a počtu provedených transakcí.

Klíčová slova veřejné API, Uniqway, návrh API, monetizace dat, carsharing služba, API key autentizace, REST API, Java, Play framework

Abbreviations

API	Application Programming Interface
B2B	Business-To-Business
CRUD	Create, Read, Update, Delete
DAO	Data Access Object
DSL	Domain Specific Language
DTO	Data Transfer Object
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JDBC	Java Database Connectivity
JPA	Java Persistence API
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MVC	Model-View-Controller
REST	Representational State Transfer
SQL	Structured Query Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

Introduction

Sharing economics is an economic model that is based on the sharing of resources, goods, and services among individuals and organizations. The core idea of sharing economics is to increase the utilization of underutilized resources, which helps to reduce waste, increase efficiency, and promote sustainability. People can use the sharing economy in different areas, such as traveling, finances, employment, transport and many others, usually with the help of a related digital platform.

One of the areas of the sharing economy in transport, are carsharing services. These services are mainly used by occasional drivers or by people who simply cannot afford a car.

Uniqway is the first Czech student carsharing. This service is intended for students and employees of Czech universities. This project is created by students of the Czech Technical University in Prague, Czech University of Life Sciences Prague and Prague University of Economics and Business. The service is delivered to the customer by a mobile app, where any registered customer can create a reservation of any available car and rent it for a ride.

The Uniqway system collects various anonymised data on vehicle movements. Today, many other services and systems are based on knowledge of the behaviour of individuals or groups in society. This data can bring valuable information for some companies or can be used by application developers to create new innovative services, which can make the life of a customer easier.

This thesis marginally follows up to the thesis Experimental application for verifying the reliability of automatic transport mode identification of the student Filip Musal from the Faculty of Electrical Engineering, Czech Technical University in Prague. In his thesis he is dealing with implementation of an application for automatic detection of user transport mode. One of the outputs of his thesis is an android library for transport mode detection. This library will be implemented in the future into the Uniqway mobile application which will allow the system to acquire anonymised transport mode data from mobile phones of users.

The aim of the work is to prepare the Uniqway system for collecting data from users' mobile applications, then to analyze the relevant data that the system collects about the movement of users and vehicles, and to design a public interface within which this data will be exposed. Another goal of this thesis is to propose how the exposed data could be monetized in the future. The implementation part of this work is solved only within the backend part of the Uniqway system.

This thesis is divided into four chapters. In the first chapter I deal with the theory, which is necessary for the practical part of the thesis. The next chapter contains the requirements analysis and solution design which is implemented in the following implementation chapter. The last chapter deals with data monetization.

Chapter 1
Theoretical part

In this chapter I cover necessary theory which I will use in the following chapters, mostly during analysis and solution design.

In the first section, I write about application programming interface – what it is and what it is used for, the second section is focused on principles of REST API. The next section is focused on the basics of API security, which is an integral part of any API implementation. The last section describes technologies the Uniqway system uses.

1.1 Application programming interface

An application programming interface (API) *"is a set of defined rules that enable different applications to communicate with each other"*. [1]

The interface can be also imagined as some kind of a contract of service between two software systems. Two subjects communicating to each other through some API are usually called a client and a server. The communication takes place in the form of sending requests and responses. Requests are sent by a client to a server and the server sends a response back to the client. The communication between client and server is illustrated in the figure 1.1.

An integral part of an API is API documentation, which specifies how all the requests and responses are exactly structured. The documentation is important for software developers in order to correctly design and implement requests and responses into applications. [2]

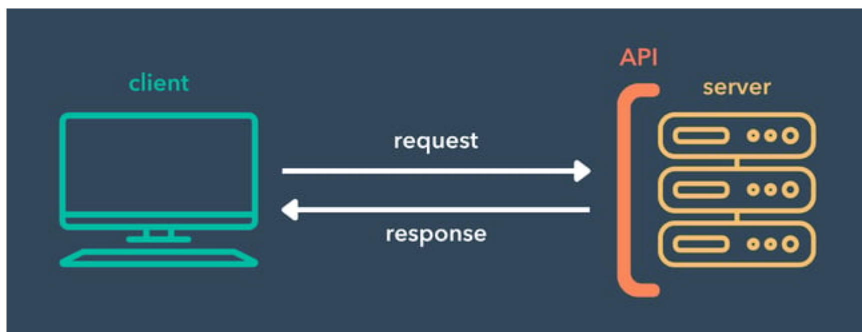


Figure 1.1 Communication of client and server through the API[3]

1.1.1 API endpoint

Another term that is used in connection with API is *endpoint*. *“API endpoints are the final touchpoints in the API communication system. These include server URLs, services, and other specific digital locations from where information is sent and received between systems.”* [2]

In other words, the endpoint is an entrypoint into the system where a client sends a request asking for some data, to thereafter receive a response containing requested data from the server.

API documentation usually contains a list of endpoints and information which data a client can request on which endpoint.

1.1.2 API types

In general, APIs can be divided into two main types – *public APIs* and *private APIs*.

Public APIs are intended to be used by a wide community of users, mostly software developers. Developers can bring new ideas of how the data of the organization can be used and build interesting and useful applications using those interfaces. Public API can help the organization as well – it can *“increase the use of company assets and add business value without direct investment in app development. Public APIs can help generate new business ideas and decrease development costs.”* On the other hand, using a public API by a lot of users can bring problems as well. Any upgrade of the API can impact all applications that are using it. Public interfaces also expose endpoints of the organization systems to the outer world, which can result in security threats. [4]

Private APIs are usually used by a restricted group of developers. It can be used by an organization’s own developers for internal integration of applications or by contracted partners for business-to-business (B2B) integration. APIs for internal purposes are known *internal APIs* and those for B2B integration are referred to as *partner APIs*. [4]

Types of APIs according to the visibility are shown in the figure 1.2.

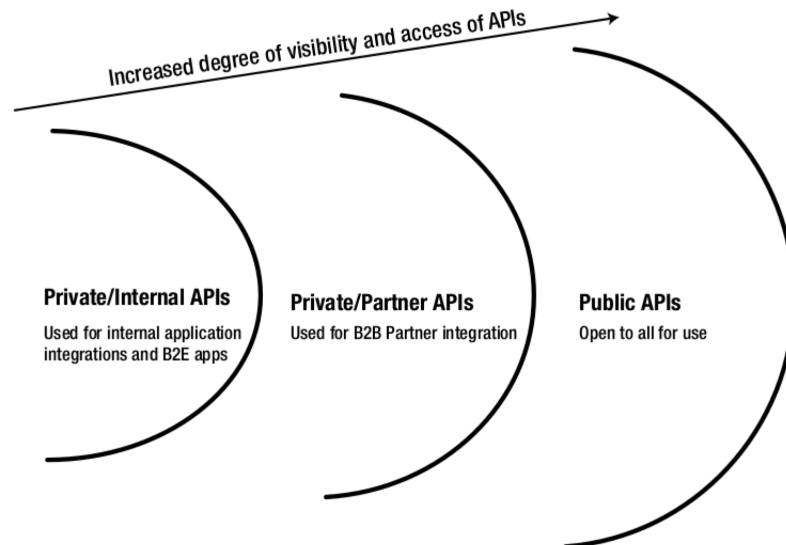


Figure 1.2 Types of APIs [5]

1.2 REST API

There are various protocols that can be used for API implementation. In this thesis, I will primarily work with REST API interfaces.

REST (Representational State Transfer) is an architectural style which sets certain guidelines and constraints to be followed during design of an API. Since an interface that follows principles of REST should be intuitive and easy to understand, it makes development of client applications simple. Implementation of REST API is usually bound to the HTTP protocol which is used for request/response sending. [6]

Main constraints for building a REST application are [7]:

- Uniform Interface
- Client-Server
- Stateless
- Cache
- Layered System
- Code-On-Demand

REST emphasizes on a *Uniform interface* between components. It simplifies the overall system architecture and improves the visibility of interactions. Since the data are sent in a standardized way, it also helps to decouple the individual components. Another constraint is a *Client-Server* architectural style. This style follows the separation of concerns principle. It allows separate development of server and client applications. Therefore, the client applications can use different technologies, programming languages and run on different platforms. *Stateless* constraint states that the communication between client and server must be stateless. Each request sent to the server by a client should contain all necessary information so the server understands it and returns expected response. Hence, the state of the session is completely handled by the client application and the server does not have to keep any context of the communication. *Cache* constraint improves network efficiency. Request and response data are labeled as cacheable or non-cacheable. *"If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests."* Main advantage of this constraint is that it can reduce latency by partially or completely eliminating some of the interactions. The *Layered system* constraint states that the architecture of the system is composed of layers. A component belonging to one layer should not know about components beyond the immediate layer it interacts with. The layer constraint reduces the overall system complexity. *"Layers can be used to encapsulate legacy services and to protect new services from legacy clients, simplifying components by moving infrequently used functionality to a shared intermediary."* Intermediaries can be also used for example for load balancing which improves scalability of the system. *Code-on-demand* is an optional constraint. It allows extension of client functionality by downloading code from the server in the form of applets or scripts. It can simplify client applications by reducing a number of pre-implemented features. [7]

1.2.1 Resource identification

In the context of a uniform interface, web services primarily work with *resources. "A resource is any web-based concept that can be referenced by a unique identifier and manipulated via the uniform interface."* Single resource usually represents an entity or collection of entities from our problem domain. Any information the server is able to provide may be considered as a resource. [8]

In the context of the Uniqway system, a resource can be a car, a collection of cars, user, reservation, invoice, payment, ride and so on.

Resources are identified by a URI (Uniform Resource Identifier). The definition of a URI according to the RFC 3986 specification is: *"A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource."* [9]

The structure of URI looks the following [9]:

```
scheme ":" ["//" authority] ["/"] path [ "?" query ] [ "#" fragment ]
```

- **Scheme** – *"Each URI begins with a scheme name that refers to a specification for assigning identifiers within that scheme,... scheme's specification may further restrict the syntax and semantics of identifiers using that scheme"*
- **Authority** – *"URI schemes include a hierarchical element for a naming authority so that governance of the name space defined by the remainder of the URI is delegated to that authority"*, in our case, the authority is the hostname of the server application
- **Path** – *"The path component contains data, usually organized in hierarchical form, that, along with data in the non-hierarchical query component, serves to identify a resource within the scope of the URI's scheme and naming authority"*
- **Query** – The query component contains additional, non-hierarchical data to identify a resource, often in the form of key-value pairs
- **Fragment** – *" The fragment identifier component of a URI allows indirect identification of a secondary resource by reference to a primary resource and additional identifying information"*

Here in an example of the URI which identifies a resource *users*:

```
https://www.uniqway.cz/api/users
```

Let's say that this resource contains a collection of users. The client connects to the *www.uniqway.cz* address and requests the *users* resource which is stored in the location specified by the */api/users* path. The server thereafter responds with the requested collection of users or an error in case the request could not be processed correctly.

Individual resources can also be related to each other. [8] The example below shows an URI, which identifies a collection of rides of the user with *id* equal to the value of 1:

```
https://www.uniqway.cz/api/users/1/rides
```

A resource is usually stored in some database. In the response, the server does not return the database record that corresponds to the requested resource. Instead, it returns its representation. A representation of a resource is completely separated from the resource itself. A single resource can have different representations. A UI client might require representation of the resource in an HTML format but for application clients, it would probably fit better to represent it in XML or JSON format. [8]

In this thesis, I will work with the JSON format representation of resources. The following example shows JSON representation of the *user* resource:

```
{
    "email": "user@example.com",
    "firstName": "John",
    "lastName": "Smith",
    "id": 1,
    "address": {
        "street": "Street",
        "city": "City",
        "state": "State"
    },
    "university": "FIT CTU in Prague"
}
```

1.2.2 HTTP protocol

Requests and responses are usually sent via an HTTP protocol. In this work, I consider only the HTTP/1.1 version.

HTTP is a textual protocol for fetching resources. It is a client-server stateless protocol which forms the foundation of data exchange on the web. There are two types of HTTP messages – *HTTP request* and *HTTP response*.[10]

HTTP requests are composed of [11]:

- A *start-line* followed by an optional set of *HTTP headers*, collectively referred to as the *head* of the request
- A blank line
- An optional *body* of the request

A *start-line* is a single line which contains an *HTTP method* (for example GET, PUT or POST, I describe the individual methods later in the text), the *request target* which is usually the path to the requested resource and the *HTTP version.* HTTP headers specify the request or describe the included *body.* The *body* of the request contains data attached to the request. Not all HTTP methods have a body, thus it is an optional part of the request. [11]

The following example shows an HTTP request that posts a new user entity to the server:

```
POST /api/users HTTP/1.1
Host: localhost:9000
Content-Type: application/json
Content-Length: 100

{
    "email": "user@example.com",
    "firstName": "John",
    "lastName": "Smith"
}
```

The *Host* header specifies the host and the port of the service, the *Content-type* header specifies the format of the data stored in the body and the *Content-Length* header indicates the size of the body in bytes.

HTTP responses are composed of [11]:

- A *status-line* followed by a set of headers
- A blank line
- A *body* of the response

The status line contains an *HTTP version*, a *status code* that indicates success or failure of the request and a *status text* which is a brief description of the *status code*. [11]

Simplified response for the previous request may look the following:

```
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 117

{
    "id" : 1,
    "email": "user@example.com",
    "firstName": "John",
    "lastName": "Smith"
}
```

The status code of the response indicates that the request was processed correctly and the server created the requested user. Headers indicate that the server returned the data in JSON format and the length of the content of the *body* is 117 bytes. The payload of the *body* contains the created user (we can see the server assigned the *id* value equal to 1 to the user).

There are five classes of HTTP status codes. Its values are within the range of 100 to 599, inclusive. The class is determined by the first digit of the status code. The classes are following [12]:

- **1xx Informational** – interim response about the communication status, indicates the request was received and the server is processing it
- **2xx Successful** – indicates the request was successfully received, understood and accepted
- **3xx Redirection** – indicates that in order to fulfill the request, a further action needs to be performed by the client
- **4xx Client error** – indicates error in the side of the client
- **5xx Server error** – indicates the server failed to process the request

In this work, I will work with the most commonly used HTTP methods – GET, POST, PUT and DELETE. There are other methods (HEAD, OPTIONS, TRACE and CONNECT) that I will not use during the implementation part so I will not describe them further.

These verbs indicate what the client wants to do with the particular resource. They allow us to perform CRUD (create, read, update and delete) operations on the resource [13]:

- **GET** – used to retrieve information about the requested resource which is defined by the request URI, it is a read-only method hence it should never modify the data, in case of successful execution the server responds with status code of *200 OK*, in case of error, it usually responds with the *404 Not Found* or *400 Bad request* status code

- **POST** – usually used to create a new resource in the location determined by the request URI, the resource is specified in the body of the request and the response body should contain details of the created resource, in case of success the server should respond with *201 Created* status code, in case the resource could not be created, the server may respond with status code of *204 No content*
- **PUT** – usually used to update an already existing resource specified by the request URI, the updated version of the resource is stored in the request body, in case the resource was successfully updated, the server responds with the status code of *200 OK*
- **DELETE** – used to delete the resource specified by the request URI, in case of successful execution the server usually respond with the status code of *204 No content*

1.3 API security

Using a public (or partner) API is a service-to-service matter. The API is usually not used by a particular user, instead, it is mostly used by services of business organizations. The public API should contain security mechanisms to authenticate the calling service and manage access control so the service is able to use only the endpoints it is allowed to – in my case, public endpoints.

1.3.1 Authentication

"Authentication is the process of verifying whether a user is who they say they are." In practice the user authentication is usually done by authenticating claims about their identity. It is achieved by a user providing some credentials to prove that the claims are correct (for example a username is provided along with the password that is known only to the particular user). [14]

The ways of authenticating a user, also known as authentication factors, are the following [14]:

- Something you know (a secret password)
- Something you have (physical device)
- Something you are (biometric factors)

1.3.2 Access control

Access control or *authorization* is a process of control which users have access to which resources and what actions are they allowed to perform. [15]

For example, only an authenticated user is allowed to see his own detailed profile and change its attributes such as username or email address.

There are two primary approaches to access control [15]:

- **Identity-based access control** – *"first identifies the user and then determines what they can do based on who they are. A user can try to access any resource but may be denied access based on access control rules"*
- **Capability-based access control** – *"uses special tokens or keys known as capabilities to access an API. The capability itself says what operations the bearer can perform rather than who the user is. A capability both names a resource and describes the permissions on it, so a user is not able to access any resource that they do not have a capability for"*

1.3.3 API keys

As stated before, in the context of service-to-service APIs, the client is authenticated rather than end user. A service-to-service API call may occur for example between microservices within one organisation or between different organisation when a public API is exposed to access resources that have some business value. The service authentication is needed for example for billing purposes or to apply service contract limits. [16]

One of the most common forms of authentication of a service is an *API key*. *"An API key is a token that identifies a service client rather than a user."* API keys often have set an expiry time (usually months or years). To obtain access to the API, the user usually logs into a developer portal, which is operated by the organization exposing the API, and requests for an API key. The portal generates a new key and returns it to the user. The generated API key is thereafter added by the client to each request as a request parameter or a request header. [16]

The process of using an API key is also illustrated by the figure 1.3.

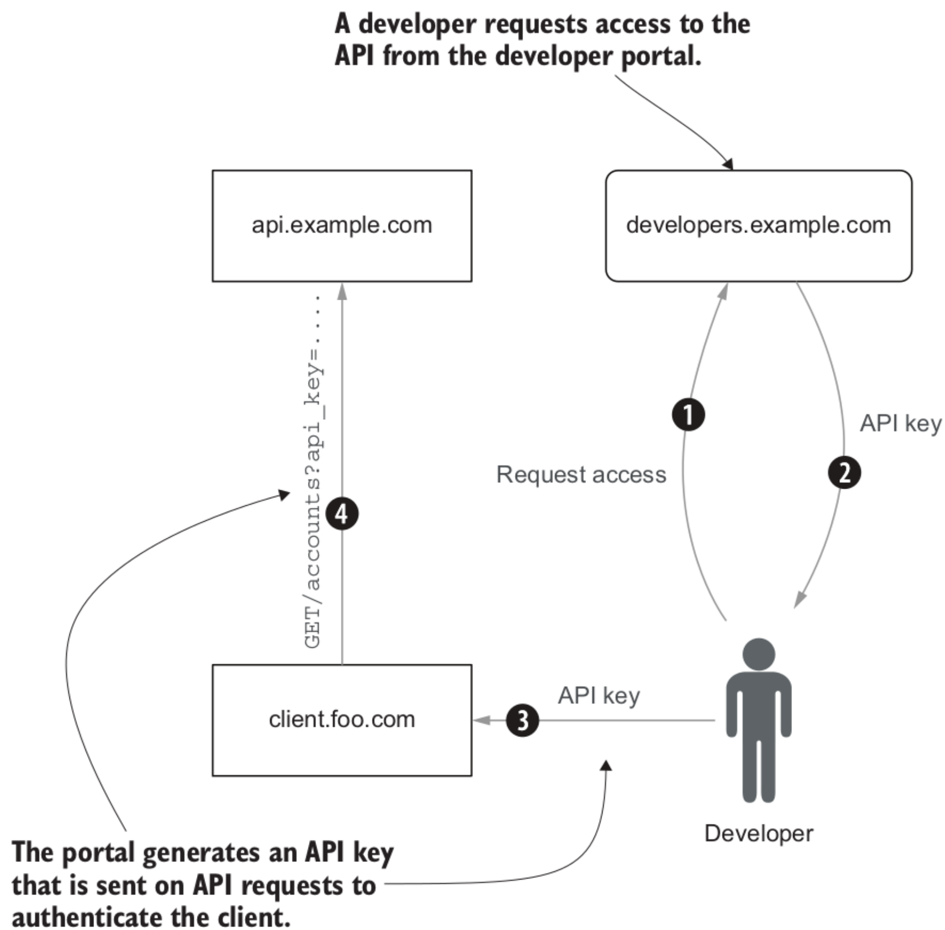


Figure 1.3 API key usage [17]

1.4 The Uniqway system technologies

In this section, I describe technologies the Uniqway system uses and which I will work with during the implementation part of my work.

1.4.1 Java Technology

Most Uniqway systems are based on a Java Technology.

The *"Java technology is both a programming language and a platform."* A platform is *"the hardware or software environment in which a program runs."* The Java platform is a *"software-only platform that runs on top of other hardware-based platforms."* The platform is illustrated by the figure 1.4. It has two components – the *Java Virtual Machine* (JVM), which is a base of java platform that is available on different hardware-based platforms, and the *Java Application Programming Interface (API)*, which is a collection of ready-made capabilities grouped into packages of related classes and interfaces. [18]

The Java programming language is a high-level language. The source code is stored in text files with *.java* extension. These files are then compiled into *.class* files by the *javac* compiler. A *.class* file then contains a *bytecode* which is a code that is not native to a particular processor, instead, it is the machine language of the Java Virtual Machine. The application is launched by the java launcher tool with an instance of the JVM. The Java Virtual Machine is available on many different operating systems so the same application can run on different platforms. [18]

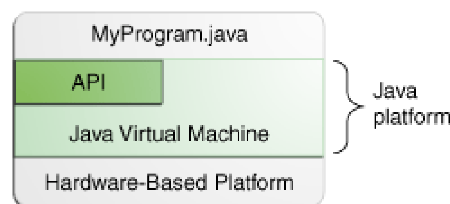


Figure 1.4 The Java platform [18]

The basic characteristics of the Java language is simplicity. The syntax is based on the C++ programming language, but the complexities of C++ are removed. It is an object oriented language which allows to build secure and portable applications. The java bytecode is interpreted and the interpreter is able to execute a program on any machine with java run-time system ported. It also provides automatic memory management – garbage collector, and supports multithreading. [19]

1.4.2 Play framework

The Uniqway backend system is written in Java programming language and a Play framework.

"Play is a high-productivity Java and Scala web application framework that integrates components and APIs for modern web application development." It is a full-stack framework which provides components to build web applications and REST services – an integrated HTTP server, form handling, routing mechanism, database schema evolution and others. The Play framework allows building web applications fulfilling a Model-View-Controller (MVC) architectural pattern. [20]

MVC is an architectural pattern that follows a *separation of concern* principle. The business logic and display of an application is separated which provides better labor division and improves maintenance. The MVC pattern consists of three parts - *Model*, *View* and *Controller*. The *model* defines data the application works with. In case the data are changed in some way, the model

usually notifies the view in order to change the display. The *view* handles layout and how the data are displayed. The *controller* processes input from users and updates the model and views in response. [21]

In the context of the Uniqway system, the *model* is represented by entity classes which are implemented in the Uniqway backend system and mapped to the data stored in the database. The *controller* is mainly represented by the service logic implemented in the backend system. The *view* logic is handled by client applications.

1.4.3 PostgreSQL

The Uniqway system uses PostgreSQL database for data storage.

PostgreSQL is an open source object-relational database system. It uses features required by the SQL standard and combines it with other features that allow us to store and scale complicated data workloads. PostgreSQL runs on all major operating systems and is highly extensible – for example allows us to define our own data types or build out custom functions. PostgreSQL also has some useful add-ons, such as PostGIS, which is described in the next section. [22]

1.4.4 PostGIS

The Uniqway system allows users to park the car only in designated parking places. These parking places must be somehow represented in the Uniqway PostgreSQL database. For this reason the PostGIS extension comes into play.

"PostGIS is a spatial database extender for PostgreSQL object-relational database. It adds support for geographic objects allowing location queries to be run in SQL." [23]

PostGIS provides the *geometry* abstract data type and its concrete subtypes which represents different geometric shapes. There are atomic subtypes of the *geometry type*, such as *Point*, *LineString*, *LinearRing*, *Polygon* and collection subtypes, such as *MultiPoint*, *MultiLineString* and *MultiPolygon*. [24]

For example, a parking place in the Uniqway system is represented by the *Polygon* data type. In this thesis, I will only work with the atomic data types [24]:

- **Point** – 0-dimensional geometry, represent single location in a coordinate space, example: *POINT (1 2)*
- **LineString** – 1-dimensional line, formed by sequence of contiguous line segments, each segment is defined by start point and end point, example: *LINESTRING (1 2, 3 4, 5 6, 7 8)*
- **LinearRing** – LineString which does not self intersect and the start point is equal to the end point, example: *LINEARRING (1 1, 2 1, 2 2, 1 2, 1 1)*
- **Polygon** – 2-dimensional planar region, delimited by exterior boundary, may contain zero or more interior boundaries, boundary is represented by the *LinearRing*, example: *POLYGON ((1 1, 2 1, 2 2, 1 2, 1 1))*

1.4.5 Python

The Uniqway backend system uses Python programming language for the backend REST API testing. Thanks to its simplicity, Python is ideal for this API testing.

Python is an interpreted, object oriented, dynamic typed programming language. It has elegant and simple syntax and effective high-level data structures which allows it to be used for fast software development and scripting. [25]

1.4.6 Elasticsearch

"Elasticsearch is the distributed search and analytics engine." It is primarily used for data indexing, searching and analysing. Elasticsearch can effectively store and index various types of data such as structured or unstructured text, numerical data and geospatial data. Thereafter the data can be searched and analyzed in near real time. The Elasticsearch can be used for various use cases, for example [26]:

- website searchbox
- store and analyze logs, metrics and security event data
- manage, analyze and integrate spatial and geographic data
- store and process genetic data

The Uniqway system uses Elasticsearch mainly for storage of analytical data, event logs of the system and car telemetry data.

"Elasticsearch is a distributed document store." It does not store data as rows of database tables, like it is done in relational databases. Instead, it stores complex data structures as documents. For representation of documents, the Elasticsearch uses JSON format. Every stored document is indexed. An index is an optimized collection of documents. A document contains individual fields – key-value pairs – which store concrete data. Elasticsearch has the capability to be schema-less which means that fields of a document do not have to have specified concrete data type. The Elasticsearch will automatically detect and map the datatype of the concrete field during indexing of the document. However, it is sometimes convenient to explicitly map individual fields to specific data types which for example enables to distinguish between full-text string fields and exact value string fields, use custom date format or use geographical data types, such as *geopoint* that cannot be automatically detected. [27]

Once the documents are indexed, there is only one thing left to do – search for data and analyze it.

Elasticsearch provides a REST API for indexing and searching the data. Within the API, the searching and indexing is performed by using the Query DSL. It is a JSON-style query language of the Elasticsearch which has support for creating structured queries, full text queries or its combination – complex queries. There is also the Elasticsearch client which allows to use the Query DSL capabilities from applications. The client is available for various languages such as Java, Python, JavaScript, .NET and others. [28]

I will present a few examples of search queries using the Elasticsearch Query DSL. Basics of these queries will be later used in the implementation part.

The following example shows a basic search query. In my case, this query retrieves all rides of cars with the model name of *Scala*, the search is performed over the *uniqway* index:

```
GET /uniqway/_search
{
    "query": {
        "match": {
            "car.model_name": "Scala"
        }
    }
}
```

Users often need to create a query which retrieves documents matching multiple criteria. For example, I want a query that retrieves all non-zero length rides of Scalas within the time interval of *2023-01-01* and *2023-01-01*. In order to do that, I have to add the *bool* clause which allows to combine different queries – in my case, it combines *match* and *range* queries. The resulting query looks the following:

```
GET /uniqway/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "car.model_name": "Scala"
          }
        },
        {
          "range": {
            "ride.start_date": {
              "gte": "2023-01-01",
              "lte": "2023-02-01"
            }
          }
        },
        {
          "range": {
            "ride.length_km": {
              "gte": 0
            }
          }
        }
      ]
    }
  }
}
```

The Elasticsearch also supports geo queries. The final example shows a geo distance query. This query retrieves all data of the *Scala* model which was collected within the distance of 1000 meters from the point with the latitude of 50.1 and longitude of 14.395. The query looks the following:

```
GET /uniqway_car_data/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "car.model_name": "Scala"
          }
        }
      ],
      "filter": {
        "geo_distance": {
          "distance": "1000m",
          "location": {
            "lat": 50.1,
            "lon": 14.395
          }
        }
      }
    }
  }
}
```

Chapter 2

Requirements analysis and solution design

This chapter deals with analysis of requirements and solution design for the creation of the public API for the Uniqway system. First, I analyze the Uniqway system architecture and the data the Uniqway system collects. Then, I create an analysis in the form of functional and non-functional requirements and build a use-case model. And finally, I state the solution design. This chapter forms the main input into the implementation part of this work.

2.1 The Uniqway system architecture

The Uniqway backend system is written in Java Play framework, as a main data storage, it uses the PostgreSQL database. Communication between the backend system and the database is internally handled by the PostgreSQL JDBC driver. The next part of the system is a storage for historical data – Elasticsearch. The backend system sends these data into Elasticsearch by using the Elasticsearch client that handles requests and responses for the Elasticsearch REST API. There is also a web application and admin application, both written in a JavaScript Nuxt.js framework. The web application is where customers can view services the Uniqway offers and register themselves. Admin application is used for the system administration. The web and admin application communicate to the system through the backend REST API.

The components I described above all run on Amazon Web Services cloud infrastructure. There are other components of AWS the system uses (for example a load balancer), but in the scope of this work I can get along only with this simplified architecture so I will not describe them.

Other Uniqway components do not run on AWS cloud. A module, which is installed into cars of the Uniqway fleet, handles car locking/unlocking and sends car telemetry data to the backend system through the REST API. Mobile applications for Android and iOS operating systems are used by end users for car reservation/ride and other services the Uniwqway offers. These applications also communicate via the REST API of the backend system.

The figure 2.1 shows a UML diagram of the Uniqway system architecture.

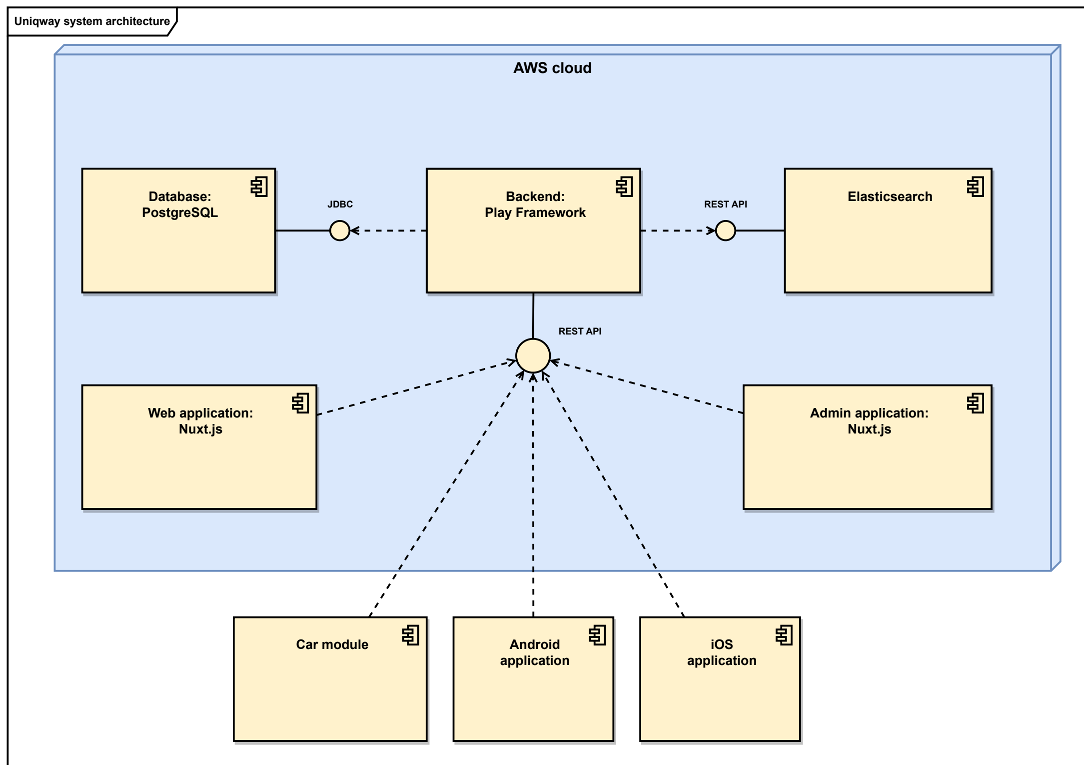


Figure 2.1 The Uniqway system architecture

2.2 The Uniqway data analysis

One of the main aims of this thesis is to create a public API which exposes relevant Uniqway data. Before I start to compile specific requirements, I perform an analysis of data the system collects and indicate which could be suitable to be exposed within the public API.

2.2.1 PostgreSQL database data

The main storage, the Uniqway backend system uses for storing production data, is the PostgreSQL database. This database stores registered users, information about the Uniqway car fleet, reservations, rides, invoices, payments and other important data for the operation of the service.

The car module collects information about the car status and sends it to the backend system which stores it into the database. These data include information about the position of the car, speed, whether it is locked/unlocked, fuel level, battery status and others. These data are sent every ten seconds to the backend which stores them into the *latest_car_data* database table. This table holds only the latest received car data, it means that once the module sends new car data, the system overwrites the old data with just received.

The data stored in the *latest_car_data* table may be relevant for the public API, it keeps current information about car status – primarily the position and speed.

2.2.2 Elasticsearch data

Elasticsearch stores mainly historical data collected from the Uniqway system for later processing by data analysts. It also stores logs from different components of the system, information about mobile application downloads, or analytical data from the mobile application. In short, data from all uniqway system components are periodically indexed into the Elasticsearch.

Data that are indexed in the Elasticsearch and could be relevant for this work are:

- **Information about rides** – contains relevant information about finished rides such as a car used for the ride, start/end location, start/end date and average speed
- **Car data** – similar data as in the before mentioned *latest_car_data* database table, but Elasticsearch does not store only the latest received data but it collects all car data a car module sends, car position data are explicitly mapped to the geopoint data type to perform geospatial queries, these queries may be useful for the public API
- **Points of interest** – information about car stops during the ongoing ride – where the car stopped and start/end date of the stop
- **Events analytics** – analytical data collected from mobile applications, these data are stored when a user uses the application to find nearest available cars for reservation, it contains location of the user at the time he sent the request, date and information about reservable cars

2.2.3 User transport mode data

As mentioned in the introduction, this thesis partially follows up on the master thesis *Experimental application for verifying the reliability of automatic transport mode identification* of the student Filip Musal from the Faculty of Electrical Engineering, Czech Technical University in Prague. In his thesis, he created a library for Uniqway android application which allows to detect a transport mode of the user – whether he goes on foot, rides a bike, travels by a vehicle or travels by a Uniqway car (as a driver or as a passenger).

This library was not implemented yet into the Uniqway android application. However, these data will be collected in the future so it is necessary to take them into account and prepare the backend system and the database for their collection. This is one of the aims of this thesis, so I am dealing with these data further in the thesis.

2.3 Requirement analysis

In this section, I state concrete requirements on the implementation of the public API. These requirements are formulated on the basis of consultation with the Uniqway student team. They are divided into functional and non-functional requirements.

This work is focused only on the backend part of the system so the requirements are created in the context of the backend implementation.

2.3.1 Functional requirements

- **F1 – API users**
  - For users of public API, the system registers a new type of user – API user
  - These users will be registered by assigning a profile of API user to already existing user
  - API users will be registered manually by administrators, the system will not provide a possibility to register profile of API user from the perspective of a common user
- **F2 – API key authentication**
  - In order to use public API, API users are authenticated by API keys attached to the request
  - System provides interface for API key generation and management, API keys are generated for concrete API users manually by administrators, database of API keys must not contain duplicate values
  - API keys may have set a validity period
  - System enables to authenticate by the API key exclusively on public endpoints, nowhere else
- **F3 – User transport mode data collection**
  - System provides interface for creation and management of different transport modes, transport modes the system stores are managed by administrators
  - System provides interface for collection of user transport mode data
  - System stores start and end date, start and end location and concrete types of collected user transport modes
- **F4 – Exposure of relevant data for the public interface**
  - System exposes relevant data the Uniqway system collects, for the public API
  - The exposed data includes information about car movement, finished rides, car stops during an ongoing ride and user transport mode
  - User is able to filter requested data by the individual attributes

2.3.2 Non-functional requirements

- **N1 – REST API interface**
  - Public API is available via REST API interface

2.4 Use case model

Use case model is a detailed specification of analyzed requirements. In this section, I state use cases derived from functional requirements from the previous section. Since this work is focused only on the backend part of the system, use cases specify concrete interactions between users (clients) and the backend. Use cases are specified on the basis of consultations with the Uniqway student team. The use case model consists of a list of actors, a list of specific use cases and a use case diagram.

2.4.1 List of actors

- **Administrator**
  - Takes care of creation and management of API users
  - Generates and manages API keys for API users
  - Creates and manages user transport mode types
- **API user**
  - Users of public API
  - Retrieves data from public API for own use
  - Authenticates by API key
- **Uniqway mobile application**
  - Uniqway mobile application used by casual users
  - Sends analytical data to the backend system

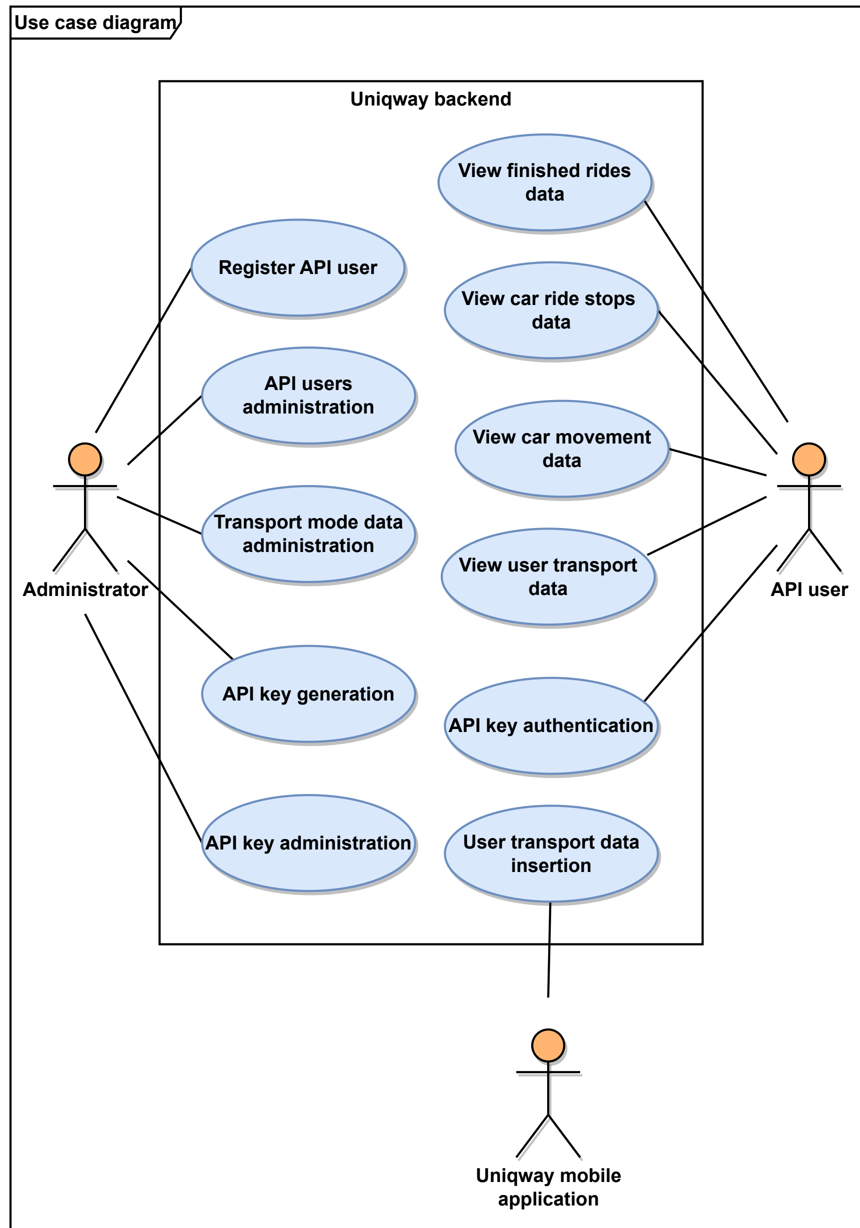
2.4.2 List of use cases

- **UC1 – Register API user**
  - Allows administrator to register new API user.
  - Administrator registers an API user manually by creating a profile of API user to an already existing user, administrator also fills in profile information
  - Profile information an administrator must fill contain information about the user it belongs to, name of a project the public API will be used for and timestamp of profile creation
- **UC2 – API users administration**
  - Allows administrator to manage API users
  - Administrator is able to view one concrete or all API users, update their profile information and delete them
- **UC3 – API key generation**
  - Allows administrator to generate new API key for a specific API user
  - To generate a new API key, administrator must enter API key information to the system
  - API key information an administrator must enter contain a specific user the API key belongs to and timestamp until which the key is valid, if the key does not have a limited validity period, the administrator does not fill it
- **UC4 – API key administration**
  - Allows administrator to manage API keys stored in the system
  - Administrator is able to view all valid API keys or one particular API key, to regenerate the value of already existing API key and to invalidate an API key
- **UC5 – API key authentication**
  - Allows client applications, using a public API, to authenticate by API key attached to the request
  - API key authentication is allowed only for public endpoints
  - If the provided API key has expired validity, the system does not authenticate the application and denies the request
- **UC6 – Transport mode data administration**
  - Allows administrator to view, store, update and delete different transport modes of users
  - To store a new transport mode, the administrator must enter the name of the mode
  - These transport modes are attached to specific data collected from mobile applications
  - For now, the system supports the following transport modes: bike riding, travelling by foot, driving a motor vehicle and carpooling within a uniqway ride
- **UC7 – User transport data insertion**
  - Uniqway mobile apps, which collect anonymised user transport data, are able to insert these data into the system
  - System is able to receive user transport data and store it, the information about these data, the system is able to store, is transport mode the particular data record belongs to, start date and end date of the transport, start location and end location of the transport

- **UC8 – View car movement data**
  - API user is able to view data of car movement
  - These data include current movement of cars on the road and historical data about car movement collected over time
  - System provides information about car identifier, car brand and model, speed, location and timestamp of when the data was received
  - API user is able to filter these data by car identifier, timestamp and is able to view the data within specified time range
  - API user is able to perform a geodistance query to retrieve data in a specific location
- **UC9 – View finished rides data**
  - API user is able to view data of finished rides collected over time
  - Information about rides, the system provides, include start and end date, start location and end location, length in kilometers, duration in minutes, car brand and model, and car fuel type
  - API user is able to filter these data by start date, end date, car brand, car model, car fuel type and is able to view the data within specified time range
- **UC10 – View car ride stops data**
  - API user is able to view data about car stops during an ongoing ride collected over time
  - Information about car stops, the system provides, include location of a stop and timestamp of locking and unlocking the car
  - API user is able to filter these data by locking timestamp and unlocking timestamp and is able to view the data within specified time range
- **UC11 – View user transport data**
  - API user is able to view user transport data collected from mobile applicatons
  - Information about user transport data, the system provides, include transport mode, start location, end location, start date and end date
  - API user is able to filter these data by start date, end date and transport mode
  - API user is able to perform a geodistance query to retrieve data in a specific location

2.4.3 Use case diagram

The use case diagram in the figure 2.2 summarizes use cases formulated in the previous section.



■ **Figure 2.2** Use case diagram

2.5 Solution design

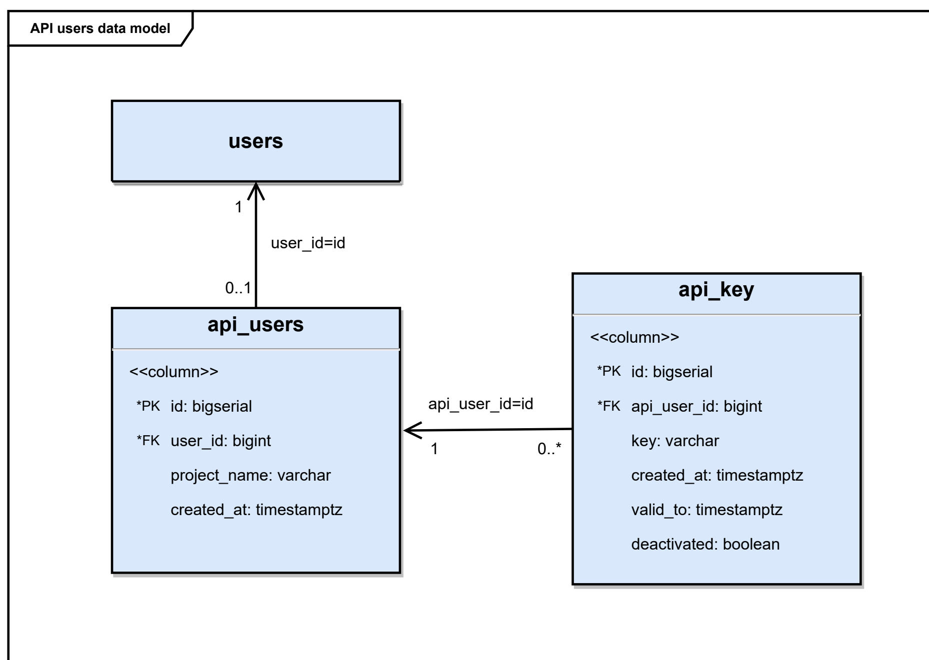
Now, I have all necessary requirements specified and I can start with the solution design. In this section I design the basic architecture of the solution. This architecture design includes a database schema design and basic design of communication between individual components of the system. The design is based on the use cases from the previous section.

2.5.1 API users and API keys

As stated in the UC1, an administrator is able to register a new API user by creating a API profile to already existing user. This profile holds a user it belongs to, name of the project and timestamp of profile creation. So the system must hold *api_users* database table which is related to the *user* table that already exists in the system.

The UC3 states that the administrator generates API keys for API users, so the system also needs to hold data of generated API keys. These keys are stored in the *api_keys* database table. This table holds the API key value, timestamp of when the key was generated, timestamp until which the key is valid and whether it was deactivated (UC4). The table is related to the *api_users* table, each API user can have zero or more keys generated.

The figure 2.3 shows UML diagram of what the solution looks like at database level.



■ **Figure 2.3** API users model

2.5.2 User transport data

Administrator is able to manage different transport modes the Uniqway system collects from mobile applications (UC6). In order to support different transport modes, the system should contain a *user_transport_modes* database table, which holds a name of the particular transport mode.

As stated in UC7, Uniqway mobile applications, that collect data about user movement, send this analytical data to the Uniqway system which must be able to store it. The user transport data is stored in the *user_transports* database table. This table is related to the *user_transport_modes* table that determines concrete transport mode. The table holds start date, end date, start location and end location of the transport mode.

The figure 2.4 shows UML diagram of what the solution looks like at database level.

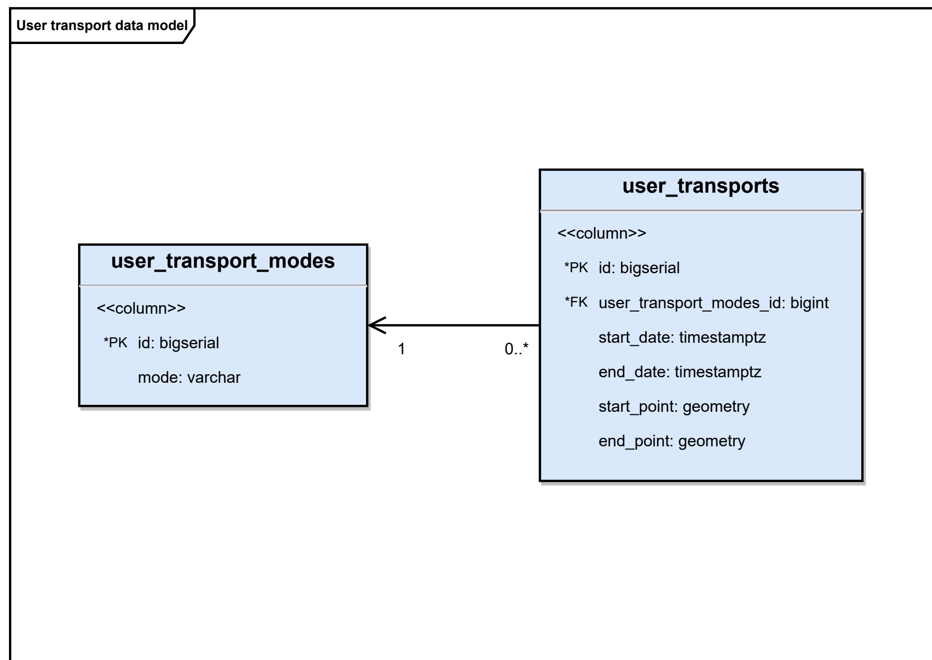


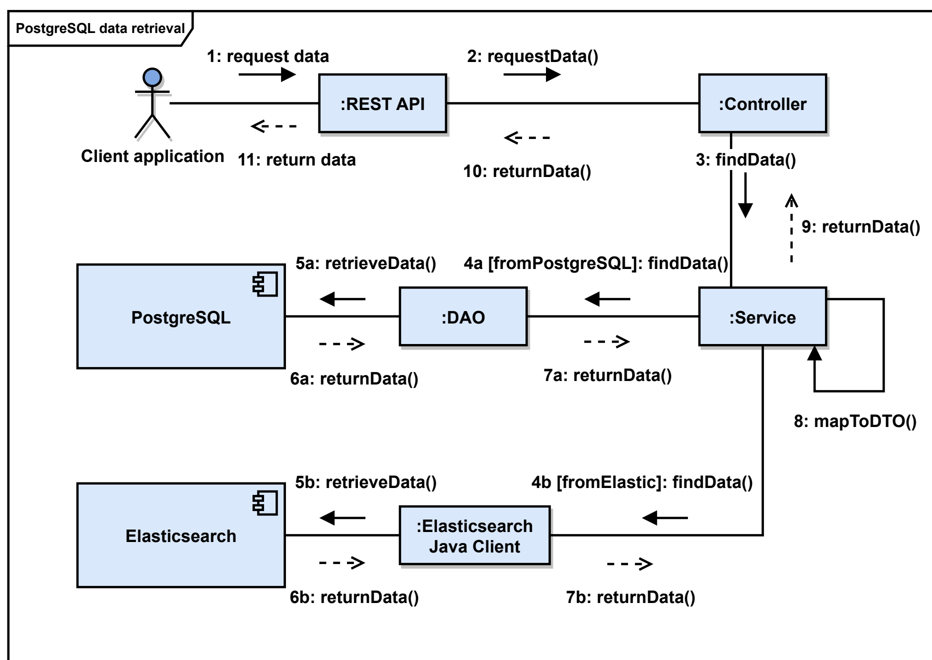
Figure 2.4 User transport data model

2.5.3 Exposure of data for public API

The data exposure itself does not require changes to the database layer. It is needed to create public REST API endpoints and implement logic that retrieves data from the Uniqway system data storage – the PostgreSQL database and Elasticsearch. I decided to divide the data into two main groups – data retrieved from the PostgreSQL database, and historical data retrieved from Elasticsearch. These groups contain the following data:

- **Data from PostgreSQL:**
  - Current movement of cars on the road (UC8)
  - User transport (UC11)
- **Data from Elasticsearch:**
  - Car movement – historical, collected over time (UC8)
  - Finished rides (UC9)
  - Car ride stops (UC10)

In case of data from the PostgreSQL, the retrieval is simple, it is only necessary to implement services that retrieve data from the database by using already implemented DAO objects and then map them to DTO objects. Methods of these services are thereafter invoked by corresponding controllers that return concrete data to the client. The retrieval of data from Elasticsearch is a little complicated. To retrieve documents from the Elasticsearch, it is necessary to implement a service that uses the Elasticsearch Java client, which is a logic that is not implemented yet in the backend. Thereafter, the service must also map the received documents to corresponding DTO objects, because they contain some data that are not intended for the public API. The service returns these DTO objects to the invoking controller. There is a high-level communication model of individual objects to retrieve data suggested in the figure 2.5.



■ **Figure 2.5** Data retrieval

Chapter 3
Implementation

In this chapter, I describe the implementation part of this thesis. The aim of this chapter is to implement use cases described in the previous chapter into the Uniqway backend system. Implementation is divided into three main parts – implementation of API users and API keys, user transport data and public API. In the end, I also describe the process of testing of implemented features.

First, I describe basic components of the Uniqway backend software project. The backend system has a three layer architecture:

- **Persistent layer** – entity classes mapped to PostgreSQL database tables (located in the *app/models/main* package), DTO classes (*app/models/dto* package) and DAO classes (*app/dao* package)
- **Controllers** – controller classes (package *app/controllers*), handles requests by client applications and returns responses
- **Service layer** – service classes (package *app/services*), main business logic of the system

There are other packages that contain additional logic, during the implementation I will work with the following:

- *app/security* – authentication, authorization classes
- *app/utils* – annotations, exceptions and other helper classes
- *app/actions* – Play framework action classes, these classes apply additional logic to incoming requests before they are passed further to the system, for example authentication of a user

The project also contains database evolution scripts located in the *conf/evolutions* directory. These scripts contain *sql* scripts of database schema evolution during the whole process of development. If a developer implements a new feature that for example includes addition of new database tables, he must write an *sql script* of database changes into a new evolution file. The system automatically applies newly added evolution scripts during start.

The Play framework also handles HTTP routing, it delegates an incoming request to the corresponding controller method. These routes, that determine the whole REST API of the backend system, are defined in the *conf/routes* and *conf/*.routes* files. In case of our system, route files are named according to the user type – *admin.routes* file defines admin routes, *client.routes* file defines routes for common logged-in users.

3.1 API users and API keys

This section describes implementation of API users, API keys, its administration and also principles and implementation of API key authentication.

3.1.1 Persistence layer

Implementation of persistence layer for API users and API keys consists of creating evolution *sql* scripts that create database tables and implementing entity and DAO classes. *Sql* scripts that create *api_users* and *api_keys* tables, according to the specification from the previous chapter, are stored in the *evolutions/134.sql* evolution file. The backend already provides base generic DAO class, so I created the *dao/main/user/ApiUserDao* and *dao/main/apikey/ApiKeyDao* classes which only extend the generic *BaseDao* class. I also implemented corresponding entity classes – *models/main/user/ApiUser* and *models/main/apikey/ApiKey*. These classes contain attributes that match columns of corresponding database tables. Attributes are mapped to columns by JPA annotations. Code 3.1 shows implementation of ApiUser class, ApiKey class is implemented similarly.

■ **Code listing 3.1** API user entity class

```
@Entity
@Table(name = "api_users")
public class ApiUser extends Model {

    @Column(nullable = false)
    private String projectName;

    @Column(nullable = false)
    private OffsetDateTime createdAt;

    @OneToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "user_id")
    private User user;

    @OneToMany(mappedBy = "apiUser")
    private List<ApiKey> apiKeys;

    // getters and setters
}
```

3.1.2 Generation of API key

During consultations with the Uniqway team, we decided that the API key would be represented by a random 16 byte hexstring. This hexstring is generated by the *utils/RandomHexStringFactory* class. For generation of random bytes, it uses the *SecureRandom* class from the Java standard library. This class provides a cryptographically strong random number generator and the *nextBytes* method that uses this generator to populate a byte array, passed as a parameter, by randomly generated bytes. Therefore, the *RandomHexStringFactory* class contains *generateRandomHexString* method, this method calls the *generateRandomBytes* method that generates random bytes by using the *SecureRandom* class. After the receival of random bytes array, the *generateRandomHexString* method converts this array to hexstring by using a *convertRandomBytesToHexstring* method and returns it. The implementation is shown in the code 3.2.

Code listing 3.2 RandomHexStringFactory class

```
@Singleton
public class RandomHexStringFactory {
    private final SecureRandom random;

    // constructor

    public String generateRandomHexString(int bytesCount) {
        byte[] randomBytes = generateRandomBytes(bytesCount);
        return convertRandomBytesToHexString(randomBytes);
    }

    private byte[] generateRandomBytes(int bytesCount) {
        byte[] bytes = new byte[bytesCount];
        random.nextBytes(bytes);
        return bytes;
    }

    private String convertRandomBytesToHexString(byte[] bytes) {
        StringBuilder stringBuilder = new StringBuilder();
        for (byte b : bytes) {
            stringBuilder.append(String.format("%02x", b));
        }
        return stringBuilder.toString();
    }
}
```

The *RandomHexStringFactory* is thereafter used in the *services/security/apikey/ApiKeyGenerateService* class which provides the *generateRandomApiKey* method (see code 3.3). This method generates an API key string, the length is defined by the *APIKEY_BYTES_COUNT* constant, in this case the constant is equal to the value of 16. Then, it checks whether this particular key already exists in the database, if not, it returns an *Optional* object containing the key, otherwise it repeats the key generation. Total number of tries is determined by the *MAX_TRIES* constant which is equal to the value of 10 in this case. If the generation is unsuccessful after a maximum number of tries, it returns an empty *Optional* object and the administrator must try it later.

Code listing 3.3 generateRandomApiKey method

```
public Optional<String> generateRandomApiKey() {
    for (int i = 0; i < MAX_TRIES; i++) {
        String key = randomHexStringFactory
                        .generateRandomHexString(APIKEY_BYTES_COUNT);
        if (!alreadyExists(key)) {
            return Optional.of(key);
        }
    }
     return Optional.empty();
}
```

3.1.3 Services

I implemented *services/security/apikey/ApiKeyService* class that provides *getValidApiKeys* method that finds all valid API keys, *getByKey* and *getById* methods for searching of one concrete API key and *get* method that maps an *ApiKey* entity class to DTO class – *GetApiKeyDto*. This service also contains *resetApiKey* method which resets the key value of already existing API key, *delete* method which deactivates API key (sets the *deactivated* attribute to true) and *generateApiKey* method that uses *ApiKeyGenerateService* to generate a new key for particular user. Part of this method is shown by the code 3.4 – the key must be generated in a *synchronized* block of code to ensure that only one thread is generating the key, so there is no key duplicity conflict caused by concurrent access to the database.

Administration of API users is handled by the *services/admin/user/ApiUserService* class. This class provides create, read, update and delete methods for ApiUser entities and also provides the *generateApiKeyForApiUser* method that allows administrators to generate API keys for a particular user.

Code listing 3.4 API key is generated in a synchronized block of code

```
...
ApiKey apiKey = new ApiKey();
synchronized(this) {
    Optional<String> key = apiKeyGenerateService.generateRandomApiKey();

    if (key.isEmpty()) {
        throw new UnableToGenerateApiKeyException();
    }

    apiKey.setKey(key.get());
    apiKey.setCreatedAt(dateTimeProvider.offsetDateTimeNow());
    apiKey.setValidTo(validTo);
    apiKey.setApiUser(apiUser);
    persist(apiKey);
    return get(apiKey);
}
```

3.1.4 API key authentication

Authentication is handled by the *actions/ApiKeyAuthenticatorAction* class. This class extends the Play framework abstract class *Action* that handles incoming requests to the server (as stated in the introduction of this chapter). A class which extends *Action* must override a *call* method that executes the action with HTTP request passed as a parameter. The code 3.5 shows implementation of the *call* method by the ApiKeyAuthenticatorAction class. This method handles the authentication; first it checks whether the key is present in the request header, if not, it returns a response indicating that the API key is missing. Then, it finds the provided key in the database and checks whether it is valid, if the key is not found in the database or is not valid, it returns a response indicating that the API key is invalid, otherwise it delegates the request to the wrapped action.

During the authentication, the *call* method uses the *security/ApiKeyAuthenticator* class. This class provides helper methods for the authentication such as *getApiKeyFromRequest* that retrieves the key from the request header, *findApiKey* that finds API key by a key string value in the database by using the *ApiKeyService*, and methods that build responses in case of failure.

This *Action* class is used by annotating corresponding controllers with the *@With(ApiKeyAuthenticatorAction.class)* annotation, provided by the Play framework, which wraps controller actions by the action class passed as a parameter.

Code listing 3.5 ApiKeyAuthenticatorAction call method

```
@Override
public CompletionStage<Result> call(Http.Request req) {
    Optional<String> key = apiKeyAuthenticator.getApiKeyFromRequest(req);
    if (key.isEmpty()) {
        return CompletableFuture
                    .supplyAsync(
                        apiKeyAuthenticator::missingApiKeyResponse);
    }

    Optional<ApiKey> apiKey = apiKeyAuthenticator.findApiKey(key.get());
    if (apiKey.isEmpty()) {
        return CompletableFuture
                    .supplyAsync(
                        apiKeyAuthenticator::invalidApiKeyResponse);
    }

    if (!apiKeyAuthenticator.apiKeyValid(apiKey.get())) {
        return CompletableFuture
                    .supplyAsync(
                        apiKeyAuthenticator::invalidApiKeyResponse);
    }

    return delegate.call(req);
}
```

3.1.5 Controllers and admin API

Controller classes *controllers/admin/ApiUserController* and *controllers/security/ApiKeyController* handle requests for administration of API users and API keys. *ApiUsersController* class contains methods for handling find, create, update and delete requests and also *generateApiKeyForApiUser* method for handling requests for generating a new API key for a particular user. *ApiKeyController* class provides methods for handling find requests and also requests for deactivating an API key and reset an existing key.

All implemented logic of managing API users and API keys is wrapped by the administrator REST API created for the admin application. Simplified description of the API is the following:

- **GET /api-user**
  - Description: Finds all API users
  - Response:
    - 200 OK
    - List of *GetApiUserDto* JSON objects
- **GET /api-user/{id}**
  - Description: Finds API user by id provided by the *id* path variable
  - Response:
    - 200 OK
    - *GetApiUserDto* JSON object

- **POST /api-user**
  - Description: Creates new API user
  - Request:
    * *StoreApiUserDto* JSON object
  - Response:
    * 201 Created
    * *GetApiUserDto* JSON object
- **POST /api-user/apikey/generate**
  - Description: Generates new API key for API user
  - Request:
    * *GenerateApiKeyDto* JSON object
  - Response:
    * 201 Created
    * *GetApiKeyDto* JSON object
- **PUT /api-user/{id}**
  - Description: Updates API user determined by the *id* path variable
  - Request:
    * *UpdateApiUserDto* JSON object
  - Response:
    * 200 OK
    * *GetApiUserDto* JSON object
- **DELETE /api-user/{id}**
  - Description: Deletes API user determined by the *id* path variable
  - Response:
    * 204 No Content
- **GET /apikey**
  - Description: Finds all valid API keys
  - Response:
    * 200 OK
    * List of *GetApiKeyDto* JSON objects
- **GET /apikey/{id}**
  - Description: Finds API key by id provided by the *id* path variable
  - Response:
    * 200 OK
    * *GetApiKeyDto* JSON object

- **PUT /apikey/{id}/reset**
  - Description: Resets key value of API key determined by the *id* path variable
  - Response:
    - 200 OK
    - *GetApiKeyDto* JSON object
- **DELETE /apikey/{id}/deactivate**
  - Description: Deactivates API key determined by the *id* path variable
  - Response:
    - 200 OK

3.2 User transport data

In this section I describe implementation of User transport modes, its collection from mobile application and administration.

3.2.1 Persistence layer

Database tables *user_transport_modes* and *user_transports*, defined in the previous chapter, are created by the *evolutions/135.sql* evolution script. This script also inserts rows into the *user_transport_modes* table to create basic transport modes: *bike*, *walk*, *motor vehicle* and *carpool*. Entity classes *models/main/public_/UserTransportMode* and *models/main/public_/UserTransport* represent these database tables in the backend.

I also implemented the *dao/main/transport/UserTransportDao* data access object class. This class provides method *findUserTransportsInGeoDistance* which returns user transports that start or end in the given distance from the given point. This method is shown in the code 3.6. It first creates a point string from parameters *lat* and *lon* that define coordination of the point. Thereafter, this point is passed into the SQL query stored in the *sql* variable. This variable represents SQL query that is used to find all records in the database from the given point in a distance determined by the *distance* parameter. To retrieve such records, the SQL query uses a PostGIS function *st_dwithin*. Because there may occur a big amount of responses, the result is paginated – the method returns a *PagedList* type. Paging parameters such as number of pages and page size are stored in the *request parameter*. This method will be used later for implementation of public API.

■ **Code listing 3.6** UserTransportDao findUserTransportsInGeoDistance method

```
public PagedList<UserTransport> findUserTransportsInGeoDistance(
                                double lat, double lon, int distance,
                                Request<UserTransport> request) {

    String pointParameter = "POINT( " + lat + " " + lon + ")";

    String sql = "select id from user_transports where
     st_dwithin(st_geogfromtext(:pointParameter), start_point, :distance)
     or st_dwithin(st_geogfromtext(:pointParameter),
     end_point, :distance)";

    RawSql rawSql = RawSqlBuilder
            .parse(sql)
            .create();

    Query<UserTransport> query = createQueryFromRequest(request);

    return query.setRawSql(rawSql)
            .setParameter("pointParameter", pointParameter)
            .setParameter("distance", distance)
            .findPagedList();
}
```

3.2.2 Services

Service classes *UserTransportModeService* and *UserTransportService* are stored in the *services/public_/transport* package. These services provide create, read, update and delete methods for UserTransportMode entities and create and read methods UserTransport entities. The *UserTransportService* also provides *getAllInGeoDistance* method, which returns records that correspond to specified area – it uses the above described method.

3.2.3 Controllers and admin API

Controller *controllers/admin/UserTransportModeController* handles administration of user transports, it contains methods for handling find, create, update and delete requests. The *controllers/public_/UserTransportController* controller class contains *getAll* method for finding all transport records and *getAllInGeoDistance* method for finding all transport method in specified area. These methods will be used later during implementation of public API.

Simplified description of administrator REST API for managing user transport modes is the following:

- **GET /user-transport-modes**
  - Description: Finds all user transport modes
  - Response:
    * 200 OK
    * List of *GetUserTransportModeDto* JSON objects
- **GET /user-transport-modes/{id}**
  - Description: Finds user transport mode by id provided by the *id* path variable
  - Response:
    * 200 OK
    * *GetUserTransportModeDto* JSON object
- **POST /user-transport-modes**
  - Description: Creates new user transport mode
  - Request:
    * *UserTransportModeDto* JSON object
  - Response:
    * 201 Created
    * *GetUserTransportModeDto* JSON object
- **PUT /user-transport-modes/{id}**
  - Description: Updates user transport mode determined by the *id* path variable
  - Request:
    * *UserTransportModeDto* JSON object
  - Response:
    * 200 OK
    * *GetUserTransportModeDto* JSON object
- **DELETE /user-transport-modes/{id}**
  - Description: Deletes user transport mode determined by the *id* path variable
  - Response:
    * 204 No Content

3.3 Public API

Implementation of public API does not require implementation of persistent layer because it mostly consists of data transfer from Elasticsearch, and their providing via REST API together with user transport data. Data transfer is handled by the Elasticsearch java client which is used by *services/admin/ElasticService*. This service already had an implemented logic for sending documents into Elasticsearch. I implemented the *sendSearchQueryToElastic* method into this service, which sends search query provided as a parameter into the Elasticsearch and returns results in a JSON object. JSON objects are represented by the *ObjectNode* class from java Jackson library which provides functionalities for working with JSON objects.

3.3.1 Request factory

Elasticsearch accepts queries in JSON format. In order to create custom queries, it is needed to implement functions that create these JSON queries according to the user's needs. Creation of JSON search queries is handled by the *services/public_/elastic/ElasticJsonRequestFactory* class.

This class provides methods for creating queries similar to those listed in section 1.4.6. The *createBasicJsonQueryWithFilters* method (code 3.7) creates basic query with list of filters provided as a parameter, the base of the query is as follows:

```
{
    "query" : {
        "bool" : {
            "must" : []
        }
    }
}
```

Filters are created by the following methods:

- *createJsonMatch* – creates *match* query that looks for an exact match to the given value (code 3.8)
- *createJsonRange* – creates range query 3.9, in the context of this work mostly date range queries, the *RangeRelation* parameter states, whether the attribute must be less or greater than the given value

Code listing 3.7 Creation of basic search query

```
public ObjectNode createBasicJsonQueryWithFilters(List<JsonNode> filters)
{
        return objectMapper.createObjectNode().set("query",
                    objectMapper.createObjectNode().set("bool",
                        objectMapper.createObjectNode().set(
                            "must", objectMapper.createArrayNode()
                                .addAll(filters))));
}
```

Code listing 3.8 Creation of match query

```
public <T> ObjectNode createJsonMatch(String attribute, T value) {
        ObjectNode matchNode = objectMapper.createObjectNode()
                                                .putPOJO(attribute, value);

        return objectMapper.createObjectNode().set("match", matchNode);
}
```

■ **Code listing 3.9** Creation of range query

```
public <T> ObjectNode createJsonRange(String attribute, T value,
                                        RangeRelation rangeRelation) {

        ObjectNode rangeNode = objectMapper.createObjectNode()
                                .putPOJO(rangeRelation.value, value);

        return objectMapper.createObjectNode().set("range",
                                    objectMapper.createObjectNode()
                                        .set(attribute, rangeNode));
}
```

The request factory class also provides *createGeoDistanceJsonFilter* method (code 3.10). This method creates geodistance query node, to retrieve data in the given distance from the given point.

■ **Code listing 3.10** Creation of geo query

```
public ObjectNode createGeoDistanceJsonFilter(
    String attribute, String distance, Double pointLat, Double pointLon){

        ObjectNode pointNode = objectMapper.createObjectNode()
                .put("lat", pointLat)
                .put("lon", pointLon);

        ObjectNode geoDistanceNode = objectMapper.createObjectNode()
                .put("distance", distance)
                .set(attribute, pointNode);

        return objectMapper.createObjectNode()
                                .set("geo_distance", geoDistanceNode);
    }
```

3.3.2 Pagination

Data returned in a response may contain thousands of records and such a volume of data may not fit in the memory. This problem is solved by applying a pagination, it means that the data are divided pages, where each page contains a certain number of records. The number of records on each page is usually provided by a user in a URL query parameter. The user is able to iterate through individual pages by providing a number of page he wants to retrieve (usually also provided by a query parameter). The following example shows a URL, where a user requests the system to provide the first ten records.

`https://www.example.com/data/pageLimit=10&page=1`

The Uniqway backend already supports pagination for data retrieved from the PostgreSQL database. It is managed by *pageLimit* and *page* query parameters. The pagination is processed by the *models/request/Request* class that provides parsing of pagination query parameters and thereafter it is applied by the *dao/base/BaseDao* class. Because this pagination is implemented for data retrieved from the database, it is also used for user transport data.

The backend system returns paginated data in a *PaginatedDataDto* structure containing requested data, indicator whether there exists another page, total number of pages and total number of records. The structure looks the following:

```
{
    "data": [],
    "meta": {
        "hasNextPage": boolean,
        "totalPageCount": number,
        "totalCount": number
    }
}
```

It is needed to implement the same logic of pagination for data retrieved from the Elasticsearch. To implement pagination, the Query DSL of the Elasticsearch provides parameters *from* and *size* that can be attached to the search query. The *from* parameter determines the number of records to be skipped from the first record, the *size parameter* specifies the page size. The following example shows a search query where a user requests first ten records of *Scala* car model from the *uniqway* index:

```
GET /uniqway/_search
{
    "query": {
        "match": {
            "car.model_name": "Scala"
        }
    },
    "from": 1,
    "size": 10
}
```

I implemented the pagination of data from the Elasticsearch into the *services/public_/elastic-/ElasticPaginationService*. This service provides the following methods:

- *applyPaginationForSearchRequest* – attaches the *from* and *size* parameters to the request so the Elasticsearch returns paginated data
- *getPaginatedData* – maps already returned paginated data to the above metioned *PaginatedDataDto* structure

3.3.3 Filters

The Uniqway backend system also provides client applications with the ability to filter data by custom filters. These filters can be defined in a URL query parameter *filters*. Each filter has a name corresponding to the name of the attribute the entity is filtered by, followed by a colon symbol, and a value. Individual filters are separated by a comma symbol. The following URL example shows a request for list of users filtered by a value of the first name and the last name of a user.

```
https://wwww.uniqway.cz/api/admin/users/filters=profile.firstName:John,
                                                profile.lastName:Smith
```

Requested entity, to which the client wants to apply custom filters, must implement the *models/request/Filterable* interface. This interface contains a *getFilters* method the entity to filter must implement that should return a map of all available filters. Filters are applied by the *dao/base/BaseDao* class. The logic of filters is used to filter *UserTransport entities*. The code 3.11 shows *getFilters* method of *UserTransport* class.

Code listing 3.11 UserTransport getFilters method

```
@Override
public Map<String, Pair<Filter, Class>> getFilters() {
    Map<String, Pair<Filter, Class>> filters = new HashMap<>();
    filters.put("startDate", new Pair<>(Filter.LIKE_DATE, String.class));
    filters.put("endDate", new Pair<>(Filter.LIKE_DATE, String.class));
    filters.put("userTransportMode.name", new Pair<>(Filter.EQUALS,
                                                  String.class));
    return filters;
}
```

As it was in case of pagination, the filters also work only for data requested from the PostgreSQL database, thus it is needed to implement the logic of processing of filters also for data retrieved from the Elasticsearch. The *Filterable* interface is implemented by DTO classes stored in the *models/dto/public_* package that are used for returning data from the Elasticsearch. I implemented the *ElasticFilterSortApplyService* class that handles the logic of creating list of custom query filters. It uses methods for creating match queries and range queries from the *ElasticJsonRequestFactory* class (see the section 3.3.1). The code 3.12 shows implementation of *applyFiltersForSearchRequest* that creates list of filters according to filters provided by the client. This method iterates through all the filter provided by the client, creates JSON query filter according to the type of the provided filter and adds it into the list of filters to be applied during the processing of a request.

Code listing 3.12 ElasticFilterSortApplyService applyFiltersForSearchRequest method

```
public void applyFiltersForSearchRequest(List<JsonNode> filters,
                                          Request<?> request) {

    for (Map.Entry<String, Pair<Filter, Object>> entry :
                            request.getFilters().entrySet()) {
        String property = entry.getKey();
        Object value = entry.getValue().getValue();
        filters.add(requestFactory.createJsonMatch(property, value));
    }
    applyDateRangeFilterForSearchRequest(filters, request);
}
```

I also implemented the ability to filter data by date range, such a filter is defined by *dateFrom* and *dateTo* URL query parameters. These parameters should contain a name of a date field the client wants to apply the date range filter to, followed by a colon symbol, and a value the client wants to filter by. The following URL example shows a request for rides data in a date range from 2022-01-01 to 2022-02-01

`https://www.uniqway.cz/api/public/rides?dateFrom=2022-01-01&dateTo=2022-02-01`

As you can see in the code 3.12, custom date range filters are created by the *applyDateRangeFilterForSearchRequest*, the code 3.13 shows implementation of this method. This method checks whether a date range filter is present, if so, adds it into the list of filters.

Code listing 3.13 ElasticFilterSortApplyService applyDateRangeFilterForSearchRequest method

```
public void applyDateRangeFilterForSearchRequest(List<JsonNode> filters,
                                                  Request<?> request) {

 if (request.getDateFrom() != null) {
  filters.add(requestFactory.createJsonRange(
    request.getDateFrom().getKey(), request.getDateFrom().getValue(),
    ElasticJsonRequestFactory.RangeRelation.GTE));
}

 if (request.getDateTo() != null) {
  filters.add(requestFactory.createJsonRange(
    request.getDateTo().getKey(), request.getDateTo().getValue(),
    ElasticJsonRequestFactory.RangeRelation.LTE));
 }
}
```

3.3.4 Search services

The previously mentioned services for creation of requests, pagination and filters are used by *services/public_/elastic/ElasticSearchRequestFactory* class. This factory class provides a *buildSearchRequest* method that builds a *SearchRequest* object from the created JSON search query. The *SearchRequest* class is provided by the Elasticsearch Java library that represents a search request; an object of this class is passed into the Elasticsearch Java client that uses it to request data. The code 3.14 shows the implementation of the *buildSearchRequest* method that creates filters, applies pagination and creates JSON query, then, if present, it adds a qeo query, and in the end it builds the *SearchRequest* object.

Code listing 3.14 ElasticSearchRequestFactory buildSearchRequest method

```
public SearchRequest buildSearchRequest(Request request) {
    elasticFilterSortApplyService
        .applyFiltersForSearchRequest(filters, request);

    elasticPaginationService
        .applyPaginationForSearchRequest(builder, request);

    ObjectNode query = elasticJsonRequestFactory
                            .createBasicJsonQueryWithFilters(filters);

    if (geoQuery != null) {
        elasticJsonRequestFactory
            .addGeoDistanceFilterIntoJsonQuery(query, geoQuery);
    }

    String queryJson = query.toString();
    return builder.withJson(new StringReader(queryJson)).build();
}
```

The whole logic of data retrieval from the Elasticsearch are covered by services located in the *services/public_* package, where each service handles data according to use cases 8-10:

- **RideService:**
  - Retrieves non-zero length finished rides
- **CarService:**
  - Retrieves historic car movement data
  - Supports geo query to retrieve data in specific area
  - Retrieves current movement data of cars on the road
- **CarRideStopService:**
  - Retrieves historic data of car stops during an ongoing ride

The code 3.15 shows implementation of the *CarRideStopService*. The data are searched by the *findCarRideStopsFromElastic* method, within this method, data are retrieved into a DTO list. Mapping of list of hits from the response to the DTO list is handled by overloaded *get* methods that map a list of documents from Elasticsearch to a list of *GetCarRideStopDto* objects. Other search services are implemented similarly.

Code listing 3.15 CarRideStopService

```
public class CarRideStopService {
  private final String carRideStopIndexName;
  private final ElasticService elasticService;
  private final ElasticSearchRequestFactory elasticSearchRequestFactory;
  private final ElasticPaginationService elasticPaginationService;
  // constructor

  public PaginatedDataDto findCarRideStopsFromElastic(Request request) {
    elasticSearchRequestFactory
        .initSearchRequestBuilderForIndex(carRideStopIndexName);

    SearchRequest searchRequest = elasticSearchRequestFactory
                                .buildSearchRequest(request);

    SearchResponse<ObjectNode> response =
              elasticService.sendSearchQueryToElastic(searchRequest);

    List<GetCarRideStopDto> dtoList = get(response);

    return elasticPaginationService
              .getPaginatedData(dtoList, response, request);
  }

  private GetCarRideStopDto get(ObjectNode json) {
   GetCarRideStopDto getCarRideStopDto = new GetCarRideStopDto();
   getCarRideStopDto.setLocationLat(json.get("location_lat").asDouble());
   getCarRideStopDto.setLocationLon(json.get("location_lon").asDouble());
   getCarRideStopDto.setLockedDate(LocalDateTime.parse(
        json.get("locked_date").asText()).atOffset(ZoneOffset.UTC));
   getCarRideStopDto.setUnlockedDate(LocalDateTime.parse(
        json.get("unlocked_date").asText()).atOffset(ZoneOffset.UTC));
   return getCarRideStopDto;
  }
```

```
  private List<GetCarRideStopDto> get(
                  SearchResponse<ObjectNode> searchResponse) {

    return searchResponse.hits().hits().stream()
            .map(Hit::source)
            .filter(Objects::nonNull)
            .map(this::get)
            .collect(Collectors.toList());
    }
}
```

3.3.5 Controllers and API

Controllers that handle client requests for public API are stored in the *controllers/public_* package. The controller classes are *UserTransportController*, *CarController*, *RideController* and *CarRideStopController*. They are all annotated by the *@With(ApiKeyAuthenticatorAction.class)* to require an API key authentication.

A detailed documentation of public API was implemented into the Swagger documentation framework, simplified description of the final public REST API is the following:

- **GET /user-transport**
  - Description: Finds all user transport data
  - Response:
    * 200 OK
    * List of *GetUserTransportDto* JSON objects
    * Paginated
  - Filters:
    * *startDate*
    * *endDate*
    * *userTransportMode.name*
- **GET /user-transport/geo-distance**
  - Description: Finds all user transport data in the given distance from the given point
  - Response:
    * 200 OK
    * List of *GetUserTransportDto* JSON objects
    * Paginated
  - Mandatory query parameters:
    * *lat* – point latitude
    * *lon* – point longitude
    * *distance* – distance in meters
  - Filters:
    * *startDate*
    * *endDate*
    * *userTransportMode.name*

- **GET /car-ride-stops**
  - Description: Finds all car stops during an ongoing ride
  - Response:
    * 200 OK
    * List of *GetCarRideStopDto* JSON objects
    * Paginated
  - Filters:
    * Date range (*dateFrom; dateTo*)
    * *locked_date*
    * *unlocked_date*
- **GET /car-data/on the road**
  - Description: Finds all location data of current cars on the road
  - Response:
    * 200 OK
    * List of *GetCarLocationSpeedDto* JSON objects
- **GET /car-data**
  - Description: Finds all car location data
  - Response:
    * 200 OK
    * List of *GetCarLocationSpeedDto* JSON objects
    * Paginated
  - Filters:
    * Date range (*dateFrom; dateTo*)
    * *car_id*
    * *received_date*
- **GET /car-data/geo-distance**
  - Description: Finds all car location data in the given distance from the given point
  - Response:
    * 200 OK
    * List of *GetCarLocationSpeedDto* JSON objects
    * Paginated
  - Mandatory query parameters:
    * *lat* – point latitude
    * *lon* – point longitude
    * *distance* – distance in meters
  - Filters:
    * Date range (*dateFrom; dateTo*)
    * *car_id*
    * *received_date*

- **GET /rides**
  - Description: Finds all Uniqway rides
  - Response:
    * 200 OK
    * List of *GetRideDto* JSON objects
    * Paginated
  - Filters:
    * Date range (*dateFrom; dateTo*)
    * *ride_start_date*
    * *ride_end_date*
    * *car.brand_name*
    * *car.model_name*
    * *car.fuel_type*
- **GET /rides/brand/name**
  - Description: Finds all Uniqway rides for the given car brand name
  - Response:
    * 200 OK
    * List of *GetRideDto* JSON objects
    * Paginated
  - Filters:
    * Date range (*dateFrom; dateTo*)
    * *ride_start_date*
    * *ride_end_date*
    * *car.brand_name*
    * *car.model_name*
    * *car.fuel_type*

3.4 Testing

The Uniqway backend system is tested by unit tests by using a JUnit framework and API tests by using Python programming language.

3.4.1 Unit test

Unit tests are aimed at testing individual functionalities (units) separately. Components are usually tested within one software layer, if a tested unit depends on components from the lower layer, it uses mocked variants of these components. For mocking, the system uses the Mockito framework. Unit tests are located in the *test/unit* package.

I implemented unit tests that are focused on testing of correct work with API keys and correct generation of JSON queries for the Elasticsearch. Tests of individual services and their scenarios are described below:

- **ApiKeyGenerateService tests**
  - Test of correct API key generation
    - The key was successfully generated and matches a 32 characters long hexstring (shown as an example in the code 3.16)
  - Test of generating an already existing API key
    - The service returns an empty *Optional* object in case of duplicate keys
- **ApiKeyService tests**
  - Successful API key generation
    - An API key was successfully generated and stored into the database
  - Unsuccessful API key generation
    - Unsuccessfull generation due to duplicate key values, an *UnableToGenerateAPiKey* exception is thrown
    - Unsuccessfull generation due to a wrong *validTo* attibute, an *InvalidFieldValueInJsonException* exception is thrown
  - Successful API key reset
    - An API key was successfully reset and updated in the database
  - Unsuccessful API key reset
    - Unsuccessfull reset due to duplicate key values, an *UnableToGenerateAPiKey* exception is thrown
  - Successful API key generation
    - An API key was successfully deactivated, *deactivated* attribute was set to *true* and is updated in the database

- **ElasticJsonRequestFactory tests**
  - Correct match query
    * Builds correct match query according to given parameters
  - Correct range query
    * Builds correct range query according to given parameters
  - Correct geo distance query
    * Builds correct geo distance query according to given parameters
  - Correct complex query
    * Correctly builds a complex query that combines match, range and geo distance queries

Code listing 3.16 ApiKeyGenerateService unit test

```
@RunWith(MockitoJUnitRunner.class)
public class ApiKeyGenerateServiceTest {

    @Mock
    private ApiKeyDao apiKeyDaoMock;
    private ApiKeyGenerateService apiKeyGenerateService;

    @Before
    public void setUp() {
        RandomHexStringFactory randomHexStringFactory =
            new RandomHexStringFactory(new SecureRandom());
        apiKeyGenerateService =
            new ApiKeyGenerateService(randomHexStringFactory,
                                      apiKeyDaoMock);
    }

    @Test
    public void generateRandomApiKey_shouldGenerate() {
        when(apiKeyDaoMock.findByKey(any(String.class)))
                        .thenReturn(Optional.empty());
        Optional<String> key = apiKeyGenerateService
                                    .generateRandomApiKey();
        assertTrue(key.isPresent());
        assertTrue(key.get().matches("[0-9A-Fa-f]{32}"));
    }
...
}
```

3.4.2 Python API tests

API tests are aimed at complex testing of backend functionalities by sending REST API requests against a running instance of the system. Practically, these test scripts simulate client applications. Python API tests are located in the *quality-assurance* package.

I implemented API tests that are focused on testing of API key life cycle. The scenarios of these tests are the following:

- **The whole API key life cycle test:**
  - Admin created a new API user
  - Admin successfully generated a new API key to the user
  - Client is able to authenticate with the key
  - Admin successfully reset the value of the generated API key
  - Client is able to authenticate by the new value of the API key
  - Client is not able to authenticate by the old value of the API key
  - Admin successfully deactivated the API key
  - Client is not able to authenticate by the deactivated API key
- **Expired API key test:**
  - Client is not able to authenticate by an expired API key

Chapter 4

API monetization

The use of APIs has become increasingly popular in recent years. API monetization models refer to the different ways in which companies can generate revenue by providing access to their Application Programming Interfaces. The public API can be used by partners to create new interesting projects. Since the exposure of an API may bring business value to its consumers, the Uniqway public API will be monetized in the future.

One of the aims of this thesis is to propose a monetization model for the Uniqway public API. In this chapter I analyze different API monetization models, their principles and propose a suitable model for the Uniqway.

4.1 Free model

Free model is used in case a company has low-value assets and wants it to spread through different channels. Such assets may be for example a catalog of services the company provides. The free model allows consumers to use a public API without charging a fee. This model is used when an API provider wants to grow in popularity and expand to new channels to increase customer reach. [29]

4.1.1 Freemium model

The freemium model is based on the free model, but is limited in some way. This model allows consumers to use the API for free but only for a certain duration or for a certain volume of data. [29]

Concrete subtypes of such model are [29]:

- **Duration-based free model** – allows consumers to use the API for a certain duration (for example a week, month, year...)
- **Quantity-based free model** – allows consumers to use the API for a certain number of calls (for example the API owner provides 1000 API calls for free to each consumer)
- **Hybrid free model** – combination of duration and quality based models (for example the API owner provides a free access for a certain duration or until a certain number of API calls reached)

4.2 Fee-based model

Fee-based model may be used in case a company has high-value assets and consumers are willing to pay for it. This model is for example used for analytical data or payment services (the service takes a fee from each payment transaction). [29]

Variations or fee based model are [29]:

- **One-time fee** – to get an unlimited access to API, consumer pays a one-time fee for subscribing
- **Subscription fee** – consumer pays periodically (weekly, monthly...), the volume of API calls within the period may be limited
- **Pay-per-API transaction** – consumer pays for number of API transactions made
- **Pay by transaction volume** – consumer pays for volume of API calls or volume of data returned in response
- **Tiered pricing** – different bands of number of API calls are charged differently, usually, the more API calls made the less the price is (for example a price per transaction for 1 to 1000 calls monthly is higher than price per transaction for 1001 to 2000 calls)

4.3 Revenue-sharing model

In this model, the API provider shares a part of the revenue with a partner that consumes the provider's data and exposes it further. This model is used to extend reach through various digital channels. Typical example is an advertising API. [29]

4.4 Monetization model for the Uniqway public API

In the context of the Uniqway public API, I think that there can be different variations of fee-based models for different endpoints. Endpoints providing historical data should be charged by using the *pay by transaction volume* model, as clients are likely to request larger volumes of data (e.g. vehicle movement data over a period of time) on a one-off basis. For endpoints providing in time data (movement of current cars on the road) it is better to use the *pay-per-API transaction* model because clients are likely to periodically request smaller amounts of data from such endpoints.

To implement the monetization model, a frontend interface for API users (developer portal) will need to be developed into the Uniqway system. API keys also allow to track the number of requests by a concrete project, based on this the API user can be charged.

Chapter 5
Conclusion

In this thesis, I introduced the Uniqway system and analyzed the data that the Uniqway system collects. Then, I selected relevant data that can be used to implement a public interface. Furthermore, I prepared the backend of the Uniqway system to collect data from the mobile applications of the users. The selected data was exposed in the public interface which was implemented in the form of REST API.

The backend system now also supports the creation of API users that can use the public API. For these users, an authentication mechanism using API keys has been implemented as a part of this work.

Finally, I have analyzed the monetization models for the exposed data. It turns out that models based on the number of transactions and the volume of data provided are the best fit for the Uniqway system.

During this work, I collaborated with the Uniqway student team. This thesis successfully met all the objectives set out in the assignment. I have implemented an extensible basis for exposing data to the Uniqway system, which can be built upon in future work that will, for example, deal with a frontend application for API users. Extending this work depends on the data that the system collects. In the future, for example, cars might collect additional data that could be included in the public API. This data may include, for example, weather data or road events data.

I see possible improvements to this work in the management of API keys, it would be preferable in the future to implement a more secure way of storing and sending them over the network.

Bibliography

1. *What is an API (application programming interface)?* [online]. IBM [visited on 2023-04-17]. Available from: https://www.ibm.com/topics/api.
2. *What is an API (application programming interface)?* [online]. Amazon Web Services [visited on 2023-04-17]. Available from: https://aws.amazon.com/what-is/api/.
3. What is an API endpoint? In: *HubSpot* [online]. Jamie Juviler, 2022 [visited on 2023-04-17]. Available from: https://blog.hubspot.com/website/api-endpoint.
4. DE, Brajesh. Introduction to APIs. In: *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. First edition. Bangalore, Karnataka, India: Apress, 2017, pp. 6–8. ISBN 978-1-4842-1306-3.
5. DE, Brajesh. Types of APIs. In: *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. First edition. Bangalore, Karnataka, India: Apress, 2017, p. 7. ISBN 978-1-4842-1306-3.
6. DE, Brajesh. Designing a RESTful API Interface. In: *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. First edition. Bangalore, Karnataka, India: Apress, 2017, pp. 29–30. ISBN 978-1-4842-1306-3.
7. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures* [online]. 2000. [visited on 2023-04-18]. Available from: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. Dissertation. University of California, Irvine.
8. DE, Brajesh. Designing a RESTful API Interface. In: *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. First edition. Bangalore, Karnataka, India: Apress, 2017, pp. 31–33. ISBN 978-1-4842-1306-3.
9. *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax* [online]. RFC Editor [visited on 2023-04-19]. Available from: https://www.rfc-editor.org/rfc/rfc3986.html.
10. *An overview of HTTP* [online]. MDN Web Docs [visited on 2023-04-20]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.
11. *HTTP Messages* [online]. MDN Web Docs [visited on 2023-04-20]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages.
12. *RFC 9110: HTTP Semantics* [online]. RFC Editor [visited on 2023-04-23]. Available from: https://www.rfc-editor.org/rfc/rfc9110.html#name-status-codes.
13. DE, Brajesh. Designing a RESTful API Interface. In: *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. First edition. Bangalore, Karnataka, India: Apress, 2017, pp. 37–40. ISBN 978-1-4842-1306-3.

14. MADDEN, Neil. What is API security? In: *API Security in Action*. Shelter Island, NY: Manning Publications, 2020, p. 21. ISBN 9781617296024.
15. MADDEN, Neil. What is API security? In: *API Security in Action*. Shelter Island, NY: Manning Publications, 2020, p. 22. ISBN 9781617296024.
16. MADDEN, Neil. Securing service-to-service APIs. In: *API Security in Action*. Shelter Island, NY: Manning Publications, 2020, pp. 383–385. ISBN 9781617296024.
17. MADDEN, Neil. API keys and JWT bearer authentication. In: *API Security in Action*. Shelter Island, NY: Manning Publications, 2020, p. 384. ISBN 9781617296024.
18. *About the Java Technology* [online]. Oracle [visited on 2023-04-25]. Available from: `https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html`.
19. GOSLING, James; MCGILTON, Henry. Introduction to Java Technology. In: *The Java Language Environment* [online]. Oracle, 1996 [visited on 2023-04-25]. Available from: `https://www.oracle.com/java/technologies/introduction-to-java.html#318`.
20. *What is Play?* [online]. Play Framework [visited on 2023-04-25]. Available from: `https://www.playframework.com/documentation/2.8.x/Introduction`.
21. *MVC* [online]. MDN Web Docs [visited on 2023-04-25]. Available from: `https://developer.mozilla.org/en-US/docs/Glossary/MVC`.
22. *PostgreSQL: About* [online]. The PostgreSQL Global Development Group [visited on 2023-04-25]. Available from: `https://www.postgresql.org/about/`.
23. *About PostGIS* [online]. POSTGIS [visited on 2023-04-25]. Available from: `https://postgis.net/`.
24. *Chapter 4. Data Management* [online]. POSTGIS [visited on 2023-04-25]. Available from: `https://postgis.net/docs/manual-3.3/using_postgis_dbmanagement.html`.
25. *The Python Tutorial* [online]. Python Software Foundation [visited on 2023-04-25]. Available from: `https://docs.python.org/3/tutorial/index.html`.
26. *What is Elasticsearch?* [online]. Elasticsearch [visited on 2023-04-25]. Available from: `https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html`.
27. *Data in: documents and indices* [online]. Elasticsearch [visited on 2023-04-25]. Available from: `https://www.elastic.co/guide/en/elasticsearch/reference/current/documents-indices.html`.
28. *Information out: search and analyze* [online]. Elasticsearch [visited on 2023-04-25]. Available from: `https://www.elastic.co/guide/en/elasticsearch/reference/current/documents-indices.html`.
29. DE, Brajesh. API Monetization. In: *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. First edition. Bangalore, Karnataka, India: Apress, 2017, pp. 143–152. ISBN 978-1-4842-1306-3.

Obsah přiloženého média

```
readme.txt ........................ stručný popis obsahu média
src
└── thesis ........................ zdrojová forma práce ve formátu LaTeX
text .............................. text práce
└── thesis.pdf .................... text práce ve formátu PDF
```