Assignment of master's thesis

Title:	Personalised real estate search application using semantic web technologies
Student:	Bc. Tomáš Dvořák
Supervisor:	Ing. Milan Dojčinovski, Ph.D.
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Selection of a quality property is a complex process which requires domain knowledge and huge effort in reviewing a large number of properties from different real estate portals. The main goal of the thesis is to implement an application that aggregates data from various portals and provides users with advanced search capabilities. The application will be based on semantic Web technologies.

- Analyze existing real estate portals.
- Analyze existing relevant datasets (e.g. Linked Open Data datasets).
- Design and implement a server backend that will provide: data extraction from selected real estate portals using text mining methods, relevance scoring of the properties; and monitoring record changes.
- Integrate other relevant data sets (e.g. Linked Open Data data sets).
- Design and implement a front end that provides the users with preference based search capabilities. The application will display the results on a map.
- Evaluate the functionalities of the Web application.

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 21 January 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Personalised real estate search application using Semantic Web technologies

Bc. Tomáš Dvořák

Department of Web Engineering
Supervisor: Ing. Milan Dojčinovski, Ph.D.

May 5, 2022

Acknowledgements

I would like to thank my supervisor, Ing. Milan Dojčinovski, PhD, for meaningful feedback and insights at consultations. I also want to thank my partner and parents for their support in my education.

Declaration

In Prague on May 5, 2022 . . . .

Citation of this thesis

Dvořák, Tomáš. *Personalised real estate search application using Semantic Web technologies*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstract

The COVID-19 pandemic led to increasing demand for real estate, mainly for those in cities rich in civic amenities. Finding the right real estate property without any domain insights is difficult. Creating a real estate portal with more than just a base of advertisement listings can require the use of proprietary technologies, which often do not allow storing information for later usage and thus results in the state known as the vendor locking. This thesis proposes an alternative way of creating a web-based scalable application using open source technologies powered by a triple-store database which enables the potential of the linked data.

Keywords real estate, semantic web, sparql, virtuoso, react, node, nestjs, open linked data, rest api

Abstrakt

Pandemie COVID-19 vedla ke zvýšení poptávky po nemovitostech, především ve městech s bohatou občanskou vybaveností. Najít vhodnou nemovitost bez jakýchkoli znalostí realitního trhu je obtížné. Vytvoření realitního portálu s více než jen základní inzercí nabídek může vyžadovat použití proprietárních technologií, které často neumožňují ukládání informací pro pozdější využití, a tak dochází ke stavu známému jako proprietární uzamčení. Tato práce navrhuje alternativní způsob vytvoření škálovatelné webové aplikace s využitím otevřených technologií založených na databázích trojic, která umožňuje využít potenciál propojených dat.

Klíčová slova nemovitosti, sémantický web, sparql, virtuoso, react, node, nestjs, otevřená propojená data, rest api

Contents

List of Figures

List of Listings

Introduction

Due to the COVID-19 pandemic and increasing inflation, people are worried about their savings, resulting in increased demand for buying real estate [1, 2]. On the other hand side, this brings a possibility for real estate owners to sell at higher prices. The situation creates the possibility for creating a real estate portal that helps possible buyers to find quality property without just going through lots of listings pages with adverts.

Creating an application like this requires use of multiple sources for data retrieval as the user does not only want to see text with pictures but wants to be also able to filter those adverts matching their expectation. This could, for example, be finding real estate in a given city district with concrete types of amenities and transport connections. Rather than have to mention all these facts as part of their listing, real estate owners would prefer it, or might expect, that the system is able to infer them with given system knowledge. One can solve this problem with external services, which could later result in a situation called vendor locking. Another disadvantage is the **lack of possibility to store retrieved data** from external providers for subsequent usage, which forces the application to do expensive range scans and prevents developers from creating an internal knowledge base. The next issue is the **lack of semantics**, meaning that the response data has a strong structure, which is fine for developers, but it is quite limiting for the target user. Having open source technologies reduces the impact of vendor lock and, in cases with flexible but semantic data structures, unlocks possibilities to do queries across heterogeneous datasets and thus leverage the potential of satisfying more users than before. The goals of this master thesis are the following:

- analyze existing real estate portals,
- scrape and extract data from selected real estate portals,
- analyze and integrate relevant Open Linked Data sets to provide more information about scraped data,

- keeps track of all changes of each real estate,
- design own real estate portal based on semantic web technologies with regard to highly scalable and modular architecture,
- implement the web application, which provides personalized searching by a form with advanced search capabilities,
- provide testing procedures for each sub-system,
- evaluate the web application with user testing.

The thesis is organised as follows. The first chapter (**Background and related work**) introduces leading real estate portals in the Czech Republic and discusses their usability and abilities. Portals are then followed by explaining the main concepts of the Semantic web. The chapter concludes by exploring relevant open linked datasets. The second chapter (**Requirements**) defines the functional and non-functional requirements based on the thesis goals and identifies limitations of the previously mentioned real estate portals. The third chapter (**Relevant technologies**) describes which technologies have been used and compares them with their alternatives. The fourth chapter (**Design**) depicts how the project was designed and structured. The chapter also describes the communication between the individual parts. It also shows the wireframes of the web application. The fifth chapter (**Implementation**) depicts how given system parts are implemented from modules bootstrapping, dependency management, and shared database connection up to how data are received from an API. The last chapter (**Evaluation**) summarises how the whole project was tested. The chapter includes a description of unit testing, integration testing, end to end, and user testing.

Chapter 1

Background and related work

This chapter introduces leading real estate portals with their pros and cons and then explores relevant datasets and methods for creating a portal based on semantic web technologies.

1.1 Existing Real Estate portals

There are numerous real estate portals in the Czech Republic. In this Section, we will discuss the leading ones. We will mainly focus on their ability to search over thousands of adverts with selected preferences. Afterwards, we will then discuss how many property adverts the portal contains and the overall usability.

Before focusing on individual portals, it is useful to look at their overall popularity. The primary metric of popularity is the number of unique visitors; secondly, we should also look at the page views, time per session and total count of page views. All of those metrics are taken from Gemius company, the international research company, providing metrics from a webspace [3]. Gemius collects data about Czech domains for independent organisation *Sdružení pro internetový rozvoj* [3].

Table 1.1: Data metrics of explored real estate portals from November 2021 [4]

Portal	Real visitors	Page views	Total Visitors	Avg. session duration
`sreality.cz`	2 151 977	109 586 858	12 964 040	33.93 min.
`reality.idnes.cz`	918 611	60 427 816	2 446 583	16.14 min.
`bezrealitky.cz`	517 777	6 205 789	1379 076	12.12 min.
`ceskereality.cz`	230 725	2 223 178	494 643	7.88 min.

1.1.1 Sreality

The biggest and leading estate portal in the Czech Republic was founded by the company *Seznam.cz, a.s.* The portal contains over 13 000 house sale adverts and nearly 7000 flat sale adverts across the Czech Republic, where Prague represents nearly 35% of all houses and around 50% of all flats for sale. [5]

The web application's starting point is a narrow page that contains tiles of links pointing to a complex search form for a given category. Adverts are thus divided into the following categories:

- flats,
- houses,
- developer projects,
- lands,
- commercial objects,
- others - parking spaces, garages.

The complex search form mainly includes filtering based on estate properties and allows users to use filters telling more about the surrounding of a given estate. Estates properties' filter consists of the following parts.

- **Layout** multi choice (1+kk, 1+1, 2+kk, 2+1 and so forth).
- **Price** tag with lower and upper bound.
- **State** multi choice filter.
- **Extra features** multi choice for selecting extra features, like having a balcony, parking space, lift, cellar or garden.
- **Energy tag** in range A to G.
- **Floor/Usable area** with lower and upper bound.
- **Max advert age** via date picker component.
- **Floor** with lower and upper bound.
- **Ownership** (private, collective or other type of law form)
- **Structure** of given house was built of like brick, stone or wood.
- **Description search** giving text in a free form to match with description text of a given advert.

The filters that make searching through tons of listings pages with adverts more user friendly take advantage of surrounding information, and those filters are following.

- **Concrete locality** selection, one can write a specific street or city district. Search is powered by auto-completion and thus is immune to common misspelling. Users can also specify a toleration radius via the select box, which contains values varying from 0.5 km up to 25 km. The range is wide enough to accommodate the whole country and not just large cities.
- **Civic amenities** selection is composed of checkboxes representing various services like Schools, Shops, Parks or types of public transport. Users can also select the desired radius in which the amenities should be located. At the moment, the portal contains 13 types.

The whole web application is powered by the Angular framework and is rendered via CSR method. This statement can be easily verified via looking at the HTML template and seeing *ng-* prefixes and comments mentioning Angular. For map visualization the application uses a widget from `mapy.cz` (one of their next product).

Figures 1.1 and 1.2 shows preview of list a page and a detail page.

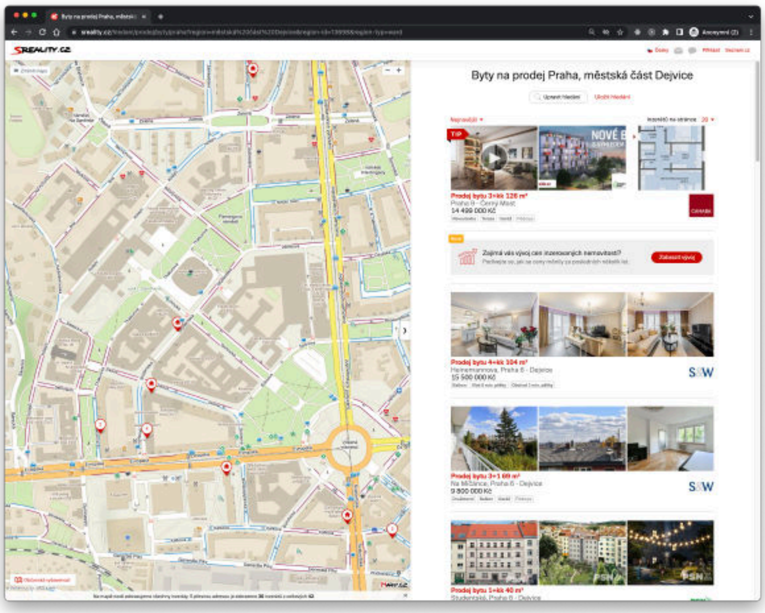


Figure 1.1: List view of adverts located in *Dejvice* district

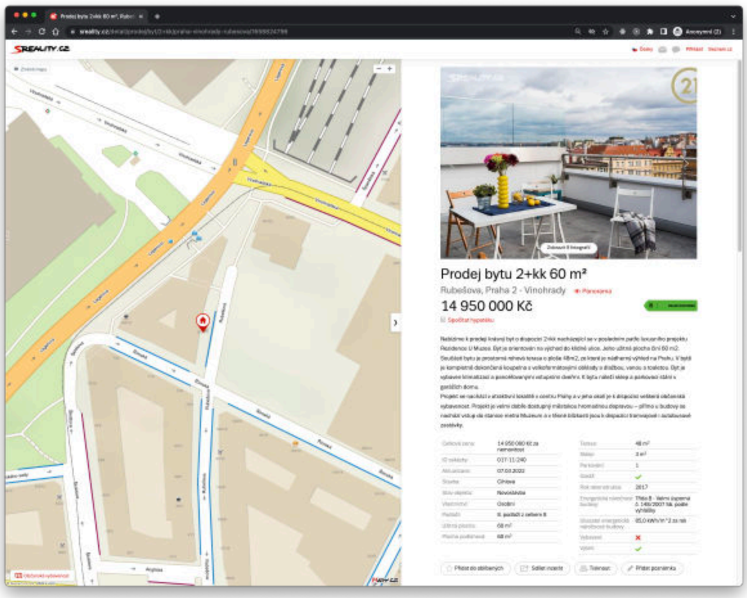


Figure 1.2: Detail view of flat for sale in *Vinohrady* district

1.1.2 Reality IDNES

The next leading estate portal with 16 000 flats and nearly 13 000 houses for sale is one of the applications belonging to MAFRA, a.s. [6]

The application firstly offers a simple search form with options selecting base category (flat, house, lands, commercial, others) and option for selecting the desired locality. The user is then able to use the advantage of more complex filters related to real estate properties. The application preserves filters in URL and thus can be easily shared. Another feature is marking adverts which were recently discounted or recently added. On the other hand, the portal does not offer any additional filter fields. In fact, the given portal does not offer any filters for near surroundings, in contrast to the *Sreality* portal.

An advert's detail page contains a description with a list of features, followed by contact details of the owner or appropriate real estate agent. The details' page also includes a small map at the end, after the contact information.

For the map visualisation, the *Open Street Map* widget is being used. From a technical point of view, the web application is implemented in PHP and based on *Nette* framework.

Figures 1.3 and 1.4 shows preview of a list page and a detail page.

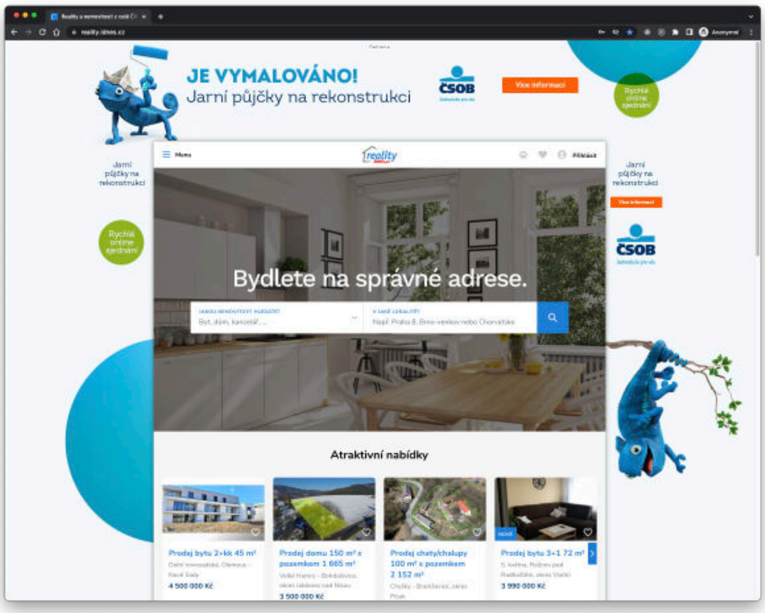
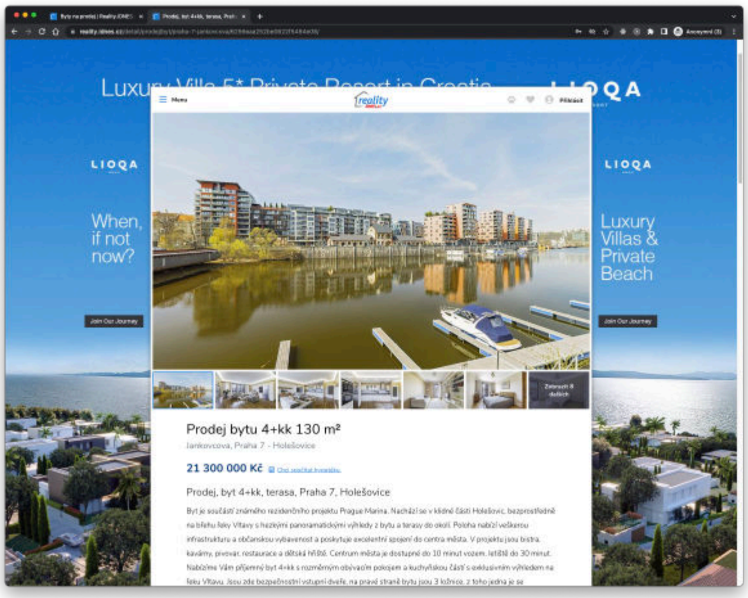

JE VYMALOVÁNO!
Jarní půjčky na rekonstrukci
ČSOB
Bydlete na správné adrese.
Atraktivní nabídky
Prodej bytu 2+kk 45 m²
4 500 000 Kč
Prodej domu 150 m² s pozemkem 1 665 m²
3 500 000 Kč
Prodej chaty/chalupy 100 m² s pozemkem 2 152 m²
Prodej bytu 3+1 72 m²
3 990 000 Kč

Figure 1.3: The generic list of adverts for the whole Czech Republic



Prodej bytu 4+kk 130 m²

Jankovcova, Praha 7 - Holešovice

21 300 000 Kč Chci spočítat hypotéku

Prodej, byt 4+kk, terasa, Praha 7, Holešovice

Byt je součástí známého rezidenčního projektu Prague Marina. Nachází se v klidné části Holešovic, bezprostředně na břehu řeky Vltavy s hezkými panoramatickými výhledy z bytu a terasy do okolí. Poloha nabízí veškerou infrastrukturu a občanskou vybavenost a poskytuje excelentní spojení do centra města. V projektu jsou bistra, kavárny, pivovar, restaurace a dětská hřiště. Centrum města je dostupné do 10 minut vozem, letiště do 30 minut. Nabízíme Vám příjemný byt 4+kk s rozměrným obývacím pokojem a kuchyňskou částí s exklusivním výhledem na řeku Vltavu. Jsou zde bezpečnostní vstupní dveře, na pravé straně bytu jsou 3 ložnice, z toho jedna je se

Figure 1.4: A detailed view of a flat for sale in the Holesovice district.

1.1.3 České Reality

This portal has a total of over 7000 flats and nearly 8000 houses for sale [7]. Expect from categories mentioned in previous portals, *ČeskéReality.cz* also focuses on advertising cottages. A further difference is the ability to browse all real estate offices with their listings and overall statistics.

The homepage starts with various tiles of categories that point to the page, beginning with a search form, followed by a list of filtered estates (by default, no filters are selected). Estate properties's filters are similar to the one mentioned before. The search form also allows filtering via estate surrounding, but in a relatively different way than the *Sreality* does. The user can select the desired public transport type via available checkboxes and specify the location of the given estate in a given district by available options like - In the centre, the outskirts of the city, busy part of the city, and so forth. For Prague, the public transport filter does not contain either tram or subway.

Figures 1.5 and 1.6 shows a preview of the list and the detailed page.

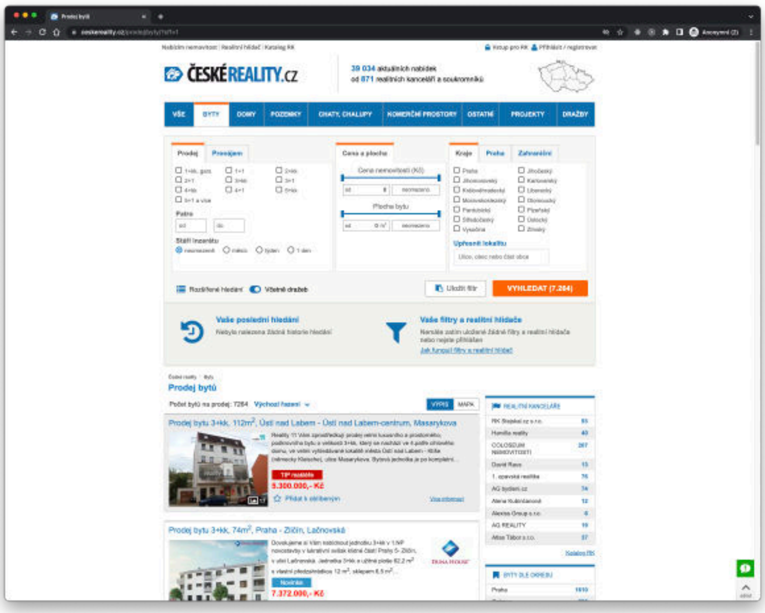

ČESKÉREALITY.cz
39 034 aktuálních nabídek
od 871 realitních kanceláří a soukromníků
VŠE
BYTY
DOMY
POZEMKY
CHATY, CHALUPY
KOMERČNÍ PROSTORY
OSTATNÍ
PROJEKTY
DRAŽBY
VYHLEDAT (7 284)
Vaše poslední hledání
Vaše filtry a realitní hlídače
Prodej bytů
Prodej bytu 3+kk, 112m², Ústí nad Labem - Ústí nad Labem-centrum, Masarykova
5.300.000,- Kč
Prodej bytu 3+kk, 74m², Praha - Zličín, Lačnovská
7.372.000,- Kč

Figure 1.5: List view of adverts located in whole Czech Republic

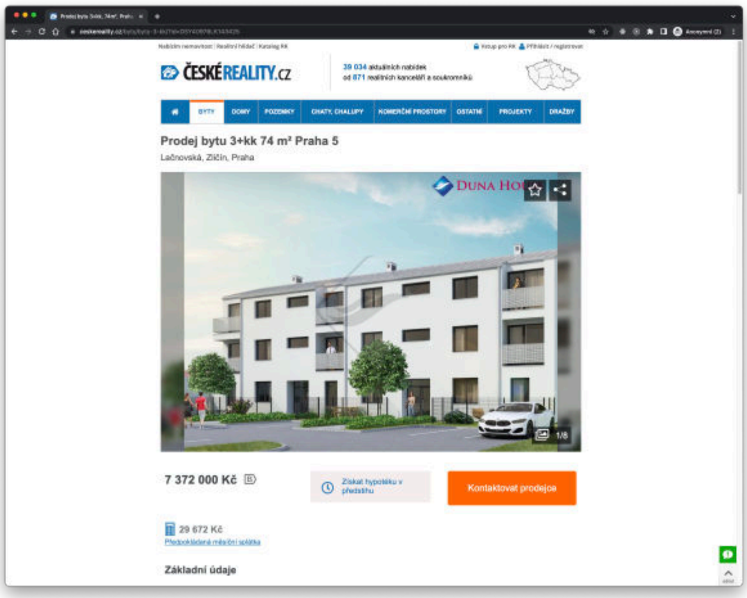


Figure 1.6: Detail view of flat for sale in Zličín district

1.1.4 Bezrealitky

What makes *BezRealitky*[1] portal different is a statement that with rising usage of the internet, people are able to sell and buy estates quickly and without the direct help of a real estate agent. This could create a situation in which both sides benefit from lower prices/agent fees. The portal states that people, thanks to using their service, save in total around 2 billion CZK yearly. [8]

Bezrealitky also focuses on those actively looking for a rental. The portal states that via creating and paying for a *Premium profile*, messages delivered to the landlord will be prioritised and should lead to higher chances of making a deal.

The homepage contains a minimalist form to filter irrelevant adverts quickly. Below the search form, text buttons with popular search tags are shown. When the filter is submitted, the result page is displayed, starting with a map on the left side and listing on the right side, similar to the look of *Sreality*. On a listing page, the user can use a more advanced filter offering the following fields.

- **Advert type** - Sell, Rent, Roommates.
- **Type of estate** - Flat, House, Commercial object, Land, Garage, Office, Atellier, Cottage.
- **Price interval** - lower and upper bound.
- **Layout** - 1+kk, 2+kk, ... 7+1.
- **Country** - Czech Republic or Slovakia followed by selecting a concrete district.
- **Ownership** - Private, Collective, General, Other.
- **Equipment** - fully, partially, none.
- **Others** - balcony, terrace, garage, lift, cellar, parking.
- **Area** - lower and upperbound in m^2.

The portal shows amenities and the closest public transport stop on an adverts detail page. For displaying adverts on a map, the *Open Street Map* widget is being used.

Figures 1.7 and 1.8 shows preview of list page and a detail page.

[1]https://www.bezrealitky.cz/

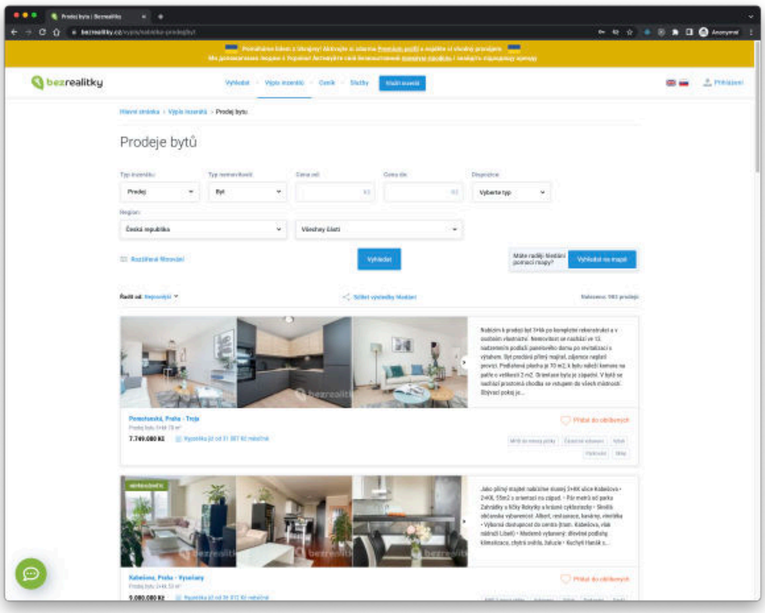


Figure 1.7: List view of adverts located in whole Czech Republic

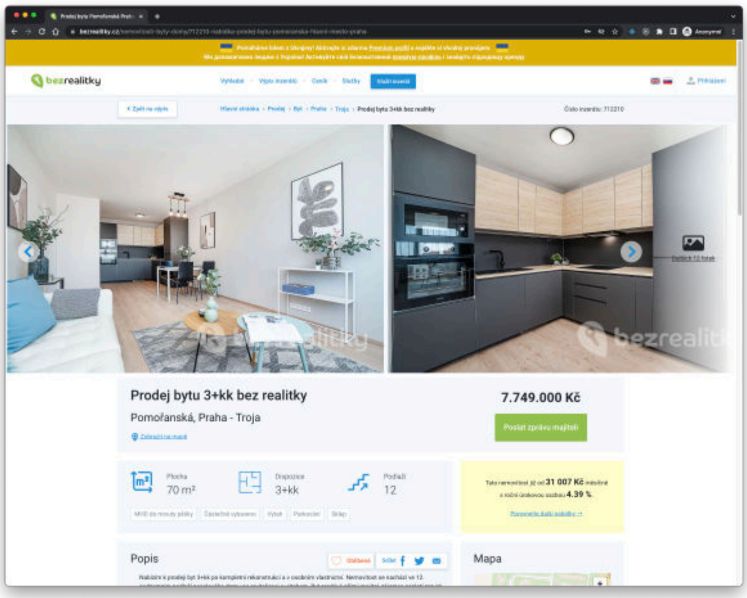


Figure 1.8: Detail view of flat for sale in Trója district

1.1.5 Summary

We have presented four major real estate portals with relatively high traffic.

From a functionality perspective, the *Sreality* portal provides the most filters and is the only portal which provides filtering by social facilities and public transport within a given distance. Expect from Reality IDNES; portals provide calculating travel time from the given advert to the user's chosen address. On the other hand, when the advert was inserted or updated is available only on *Sreality* and *České Reality.* However, on the *Sreality*, this information is influenced by *topping*[2].

From a usability perspective, the *Sreality* provides the best user experience. It is the only portal built as a SPA, visitors are automatically linked with their account on *Seznam*[3] and thus do not need any explicit login. Another is the *Bezrealitky* portal, which uses an elegant and responsive design. Marketing ads do not bother users as they are inserted between adverts, while on the detail view, they are in the form of a mortgage calculator. The same behaviour applies to *Sreality.* The next portal is *Reality IDNES*, which also has a responsive design, but the web contains the ad that wraps around the entire page content. In addition, that ad is not often related to real estate. Finally, the *České Reality* is the only a portal that does not provide a responsive design (but when the mobile user agent is detected, it redirects the user to the mobile version of the app) and also contains an ad that wraps around the entire page content. It also contains ads in the form of images on an advert's detail page.

We have described the advantages and disadvantages of particular real estate portals. A very common situation is that there is no adequate support for filtering by amenities and public transport, apart from *Sreality.* The reason why *Sreality* has strong support and connection to maps is because of their related products `mapy.cz` (maps) and `firmy.cz` (companies). Those products can or probably are interconnected and result in a competitive advantage. It is worth mentioning that using their maps widget (via `mapy.cz`), which in some way is being used for a product similar to theirs, results in a violation of *license aggrement*[4]. One can consider using services from Google because tons of APIs from maps visualisation, places search, geo-coding or finding the quickest route between two points via chosen transport. The problem remains the same as with *Sreality* or any other providers. One cannot[5] store their data; thus, no caching and no data scraping, which means every request must go through the provider API and thus be billed.

[2] paying extra fees for displaying the advert on the top of others

[3] `https://seznam.cz/`

[4] `https://licence.mapy.cz/`

[5] `https://cloud.google.com/maps-platform/terms`

The following Section will consider a different approach, using open datasets with the help of technologies from a field called the Semantic Web to enrich the overall information about the surroundings of real estate.

1.2 Semantic web

The web we use every day is known as the web of documents. Those documents are represented via HTML and then read by web browsers. HTML uses tags to represent a given piece of information, but those tags lack semantics, and thus one needs to have prior knowledge to understand the context. The Semantic Web vision is to have a web of data [9], where data have strong semantics with well-defined vocabularies, which in the ideal case are standardised and used across multiple sites. This vision supports the main idea that computers can better understand and reason about the data on the web. Semantic web are empowered by technologies such as RDF, RDFS, OWL and SPARQL [9].

1.2.1 RDF/RDFS

RDF stands for *Resource Describe Framework* and is a standard framework for representing information on the web, which both humans and computers understand [10]. The core concept is assigning statements about the described subject. The statement consists of three parts and is called **triple** [11].

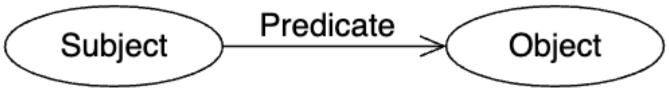


Figure 1.9: RDF Triple visualization

A set of triples is then called *RDF Graph*. Every described *subject* is uniquely identified by a given absolute IRI or blank node[6]. The *predicate* is also represented via a unique IRI. Lastly, the *object* part is one of the following types: IRI, literal[7] or a blank node.

For a better understanding, consider the following example, where we briefly describe the source located on *ex.com* domain with the use of vocabulary called *dcterms*, which defines the semantic of given predicates.

```
<http://ex.com/book/123> <http://purl.org/dc/terms/title>
    "The Merry Adventures of Robin Hood"@en .

<http://ex.com/book/123> <http://purl.org/dc/terms/description>
```

[6]Blank node refers to a local identifier used in a given RDF Graph
[7]Simple value - number, string, boolean

```
    "A book about the wildlife of Robin Hood."@en .

<http://ex.com/book/123> <http://purl.org/dc/terms/creator>
    <http://dbpedia.org/resource/Howard_Pyle_Studios> .
```

Listing 1.1: Description of a book in RDF (N-Triples serialization)

The example provided uses the simplest serialisation called *N-Triples*. One can see that the serialisation format is straight forward hence not very efficient in terms of disk size or parsing. For the following example, which is equal to the previous one, we will use a serialisation format called *Turtle*, which is an extension of the previously mentioned N-Triples [11].

```
PREFIX book: <http://ex.com/book/>
PREFIX dcterms: <http://purl.org/dc/terms/>

book:123
  dcterms:title "The Merry Adventures of Robin Hood"@en ;
  dcterms:description "A book about wild life of Robin Hood."@en ;
  dcterms:creator <http://dbpedia.org/resource/Howard_Pyle_Studios> .
```

Listing 1.2: Description of a book in RDF (Turtle serialization)

As can be seen, *Turtle* is a more compact format with the same testimonial value. To define the meanings of statements in triples, the RDFS comes in. The RDFS states for Resource description Framework Schema and is used for providing data-modelling vocabulary for RDF data [12]. RDFS introduces *Classes* and *Properties* where all of the things described by RDF are called resources [12].

1.2.2 Linked data

In the previous section, we described very briefly how the book could be described with *dcterms* vocabulary. If we look closely at *dcterms:creator* predicate, the object is not literal but an IRI. In other words, we are using a link to a given subject. This means that we are creating links between datasets, leading to a greater knowledge base. If we would specify only the author's name, we would only know the name but nothing else. With the link, we can query or look up the given link, which provides an additional set of triples about a given subject (the *RDF Graph*).

Having interlinked data means having richer information across different knowledge domains. As mentioned above, the desired goal of a Semantic Web is the ability to understand the data from both a human and computer perspective. This ability is provided via a content negotiation mechanism, where the data format of the resource is provided based on HTTP headers (concretely via *Accept* header). For the sake of simplicity, we will consider having the following source *http://dbpedia.org/resource/Howard_Pyle_Studios*, when the web browser will make a request, the following response will be in an HTML while for a reasoner (bot, scraper) that response will be in *Turtle* serialization.

For more complex relations between subjects, one can use OWL because of its richer vocabulary and ability to restrict property values. [13]

Linked Data principles

Using the following principles supports the overall interconnection of individual datasets and allows the client to browse through data similarly to how people browse through the documents on the internet. Given principles are defined as following: [14]

- use URIs as names for things,
- use HTTP URIs so that people can look up those names,
- when someone looks up a URI, provide useful information using the standards (RDF, RDFS, SPARQL),
- include links to other URIs so that they can discover additional information.

1.2.3 SPARQL

We have shown how one can create statements in triples and describe them with vocabulary. The important tool, while having data, is the ability to retrieve query them and thus retrieve meaningful information.

SPARQL is a set of specifications for querying and manipulating *RDF Graphs*. Specification of SPARQL is divided into 11 sections [15], where each section describes different areas. This section showcases the *SPARQL Query language* by itself and its important parts.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name (COUNT(?friend) AS ?count)
FROM <http://ex.com/my-graph>
WHERE {
    ?person foaf:name ?name .
    OPTIONAL { ?person foaf:knows ?friend } .
}
GROUP BY ?person ?name
ORDER BY ASC(?name) DESC(?count)
```

Listing 1.3: SPARQL query for retrieving the number of friends

Listing 1.3 returns all persons with the number of people that they know, sorted ascending by name and then descending by the number of people they know. It is essential to mention that the *WHERE* clause is required because it is the place where one sets the required graph pattern.

As been shown, SPARQL supports aggregation function as SQL does.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?s (AVG(?age) AS ?avg)
FROM <http://ex.com/my-graph>
WHERE {
    ?s foaf:age ?myAge .
    FILTER(?myAge >= 18) .

    # Usage property paths in version 1.1
    ?s foaf:knows/foaf:age ?age .
}
GROUP BY ?s
ORDER BY DESC(?avg)
```

Listing 1.4: SPARQL query for getting average age of friends

Listing 1.4 will, for every person that is at least 18, find the average age of his friends. The slash inside a query means it is a syntax form for sequence path matching [16].

The next significant feature of SPARQL is the ability to execute part of the query on another endpoint. This is fairly useful when having data linking to other data that are not within the same database[8].

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT DISTINCT ?s (group_concat(DISTINCT ?m;separator=",") as ?m)
WHERE {
  ?s a foaf:Person .

  { SERVICE <http://a.com/sparql> {?s foaf:knows/foaf:mbox ?m } }
  UNION
  { SERVICE <http://b.com/sparql> {?s foaf:knows/foaf:mbox ?m } }
  UNION
  { SERVICE SILENT <http://ex.com> {?s foaf:knows/foaf:mbox ?m } }
}
```

Listing 1.5: Example of SPAQL Federated Query

[8]RDF Database stands for the Triple Store

Listing 1.5 shows how to get a list of friends (separated with a comma) from multiple sources. The *SILENT* keywords say to the executor to continue with execution; however, the endpoint may or may not work. [17]

Here we have described how queries are constructed and what they are able to do. It is worth noting that SPARQL queries can be executed by sending an HTTP request to the given endpoint. The structure of the response is defined by standard [18].

1.3 Relevant datasets

In Section 1.1.1 we have introduced popular real estate portals in the Czech Republic. Our findings led to the conclusion that, apart from *Sreality* no other portal offers the ability to take into account civic amenities or public transport.

To take advantage of those missing features, we are looking for datasets or data sources that bring us the desired enrichment. Those data should be ideally in RDF and at least partially interlinked.

1.3.1 OpenStreetMap

OpenStreetMap[9] (OSM) is an initiative that aims to provide geographic data, such as street maps. The OSM project is mainly powered by the community. Every individual can be a contributor and helps the project to have up to date data. The contribution to the map can be made manually or automatically via additional tooling. The project states that they have more than a million volunteers worldwide. The web application of OSM provides the following features: [19]

- search for places,
- geo-coding and reverse geo-coding,
- query features around given point,
- calculating the route between two points,
- create map screenshots,
- share given map view via a link,
- view detail of given map node,
- download XML representation of the given node.

Each place node has attached a set of properties. The meaning of each property can be found in the documentation[10] page. Note, that even if link

[9] http://openstreetmap.org/
[10] https://wiki.openstreetmap.org/

is provided, it does not implicitly mean that the target provides related data in RDF format. The reason for this is simply that the source can be in a non semantic format, like CSV.

For personalized usage, OSM provides a subportal[11] where the entire datasets or weekly changesets can be downloaded. The dataset dump is provided in XML and PDB format. For using OSM in a triple store, an RDF format is needed. The conversion between provided XML/PDB dump and RDF can be made via a publicly available GitHub project called *osm2rdf*[12]. Based on the GitHub project owner, the project is maintained by the *University of Freiburg*. One can download the whole project and run the attached *Docker* image, which does the conversion. University also provides weekly dumps for the whole planet, continent or the particular country. For the Czech Republic, the dump contains nearly 135.5 million triples. Those triples are statements about almost every place in the Czech Republic.

The next project aims to provide data from the OSM in RDF is the *LinkedGeoData*. Besides the data conversion to the RDF, the project also focuses on interlinking the data to *DBpedia*. [20]

The reason why *osm2rdf* would be preferable over *Linkedgeodata* is that the vast majority of links, including those to the ontology, are not working. Besides that, the latest provided data dump is from 2015.

1.3.2 RUIÁN

The RUIÁN stands for *Register of Territorial Identification, Addresses and Real Estate*. It is a public source of addresses and territorial elements, territorial registration units and their interrelationships. [21]

Data browsing can be achieved via web[13] application or via an public SPARQL[14] endpoint. The SPARQL endpoint has been created on *Charles University Faculty of Mathematics and Physics* as part of the *Comsode*[15] project. However, the SPARQL contains outdated data because the *Czech Geodetic and Cadastral Office* do not publish data in the open linked data form.

The reason why RUIÁN is an ideal candidate for usage is that the dominant part of territorial elements in OSM comes from RUIÁN and thus are interlinked.

1.3.3 Wikidata

Wikidata stands as a free, multilingual and collaborative database, collecting structured data of its *Wikimedia* sister projects. One of the leading data

[11]https://planet.openstreetmap.org/
[12]https://github.com/ad-freiburg/osm2rdf
[13]https://vdp.cuzk.cz/
[14]https://linked.opendata.cz/sparql
[15]https://www.comsode.eu

sources for *Wikidata* is *Wikipedia*. The *Wikidata* provides its own public SPARQL endpoint, where queries can be created with the help of the query builder and thus does not need prior knowledge of Wikidata's ontology or even vocabulary. [22, 23]

Every entity in the *Wikidata* knowledge base is referenced via a persistent URL. A given entity contains exactly one label, one description, aliases and statements. Statements describe characteristics of the given entity, internally they are triples. [22]

Having a persistent URL follows recommended Linked data principles mentioned in Section 1.2.2. *Wikidata* offers for a given entity multiple data formats, whereas related links can be found in a head element in HTML response.

1.3.4 DBpedia

DBpedia is a crowd-sourced community project that creates and provides public access to their *Open Knowledge Graph* by extracting structured information from *Wikipedia*. Data is published with *Linked Open Data* principles and are available in variety of different formats like *HTML*, *Turtle*, *N-Triples*, *JSON-LD*, *XML*. [24]

The *DBpedia* similarly as *Wikidata* provides its own public SPARQL endpoint, where users can execute their queries. Because the data originally comes from *Wikipedia*, which is also the source for the *Wikidata* and because of the following *Linked Open Data* principles, one can retrieve facts about the subject by knowing only the *Wikidata* identifier. The Listing 1.6 depicts how to retrieve all triples from *DBpedia* about the *Wikidata* entity identified as *Q630893* (*Czech National Library of Technology*).

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX wikidata: <http://www.wikidata.org/entity/>

SELECT ?s ?p ?o WHERE {
   ?s ?p ?o .
   FILTER EXISTS {
      ?s owl:sameAs wikidata:Q630893
   }
}
```

Listing 1.6: Retrieve all triples about *Czech National Library of Technology* from *DBpedia* based on *Wikidata* identifier

1.3.5 Summary

We have introduced several relevant datasets with various information. The most important one is the OpenStreetMap, which provides data that can be converted to RDF via *osm2rdf* project. The *OpenStreetMap* has been chosen because it is periodically updated and can also be used as a map widget on the web application hence the data which users see on the map corresponds to those that the application uses for search. More importantly, many places are imported and linked based on that data analysis. The only disadvantage is that those data may not be provided in RDF format, but that may change in the near future. An example would be the RUIÁN, which does not provide its data in RDF, but there is a SPARQL endpoint from the third-party source which does. Data from *OpenStreetMap* often contains links to *Wikidata* which provides the public SPARQL endpoint.

The combination of OSM, *Wikidata* and RUIÁN was chosen for this thesis mainly for their timeliness and interconnectedness. The *DBpedia* dataset was omitted because all of the relevant pieces of information are already within in OSM or *Wikidata*. Based on research, there was no other relevant dataset with such a similar size.

It is worth noting that more datasets relevant to the Czech Republic can be found on *Open data portal*[16], but not all the datasets are available in RDF. To explore more datasets not relevant only to the *Czech Republic*, one can look up the *Linked Open Data Cloud*[17].

[16]https://data.gov.cz/
[17]https://lod-cloud.net/

Chapter 2

Requirements

A goal of this thesis is to create a real estate portal with advanced search capabilities based on semantic web technologies. The application should scrape data from the selected real estate portal introduced in Section 1.1 and leverage the possibilities with the use of open linked datasets. All parts of the application should be modular and thus be able to scale as the data or traffic grows with the possibilities of future extension. Users should be able to search, filter and display corresponding real estate adverts via provided UI.

2.1 Functional requirements

	Description	**Priority**
F1	**Data source:** The system will scrape, transform and store adverts from `sreality.cz` and `bezrealitky.cz`. Only flats for sale located in Prague will be included.	High
F2	**Data sync:** The system periodically revalidates stored adverts and propagates appropriate changes to its own storage. System stores history of each change that occurs on the given advert.	High
F3	**Browsing:** The application allows users to browse through all adverts via list view. Browsing should be done via a list with pagination. The user should be able to change the pagination settings.	High
F4	**Map:** The web application displays results on a map (via markers) and list. Clicking on a marker or list item will show the advert detail.	High

F5	**Detail:** A user can display the advert detail. The detail should contain information about the specific estate, including a list of local amenities and public transport nearby.	High
F6	**Search:** The application should provide personalized search abilities via the search form. The form should have, among other things, the following filter fields: • price range, • layout type, • condition, • house structure, • energy level, • ownership form, • monthly fees, • floor range, • usable area range, • type of sell (auction, classic), • added no longer than (date picker), • edited no longer than (date picker), • maximum amount of remaining annuity (only if collective ownership is selected). All form fields should be optional. The search form should provide filters for amenities and public transport within the user's desired distance. Last but not least, the user can filter adverts by providing the city district.	High
F7	**History:** A user can see adverts history with highlighted changed values over time.	High
F8	**Gallery:** A user can browse photos of selected advert.	Medium

F9	**Redirects:** The system must handle external data source advert change and prevent re-insertion of the same advert.	Medium
F10	**Homepage:** A user can see recently added adverts.	Low
F11	**Homepage:** A user can see recently updated adverts.	Low

2.2 Non-Functional requirements

	Description
N1	The system is implemented as a web platform with frontend and server services (backend).
N2	The whole system is written in a TypeScript language.
N3	Frontend is a web application powered by React library.
N4	Server related services are based on NestJS framework.
N5	API follows *json:api*[18] specification.
N6	The system should be easily extendable to use data from external providers.

[18] https://jsonapi.org/

Chapter 3

Relevant technologies

This chapter describes a set of technologies being used in the project. The technologies selection was motivated by having a single programming language (*TypeScript*) to be able to share code between sub-packages.

3.1 TypeScript

TypeScript is a strongly typed programming language that builds on top of JavaScript [25]. The typeScript was created by Microsoft and was firstly announced in October 2012 [26]. The main reason TypeScript has been adopted and gained such popularity in recent years is due to increased complexity of web application leading to increased codebase size. Using strongly typed languages provides for writing a less error-prone code. [25]

3.2 Virtuoso

Virtuoso is a high-performance and scalable Multi-Model RDBMS, Data Integration Middleware, Linked Data Deployment, and HTTP Application Server Platform. [27]

Virtuoso is currently developed as a commercial (version 8.x) and open-source version (version 7.x). For purposes of this thesis, the open-source version was chosen instead of the commercial one. One of the core functionalities that Virtuoso offers is the *RDF Triple Store* for storing triples. The Virtuoso *RDF Triple Store* is built on top of the classical relational database and transforms SPARQL queries into optimized SQL queries. [27]

Triples can be loaded into the triple store via *Virtuoso Bulk Loader* (preferred for large RDF dumps) or directly via the SPARQL endpoint, which can be accessed via HTTP protocol that conforms to the standard.

Developing an application with frequent database queries, one can decide to use a direct TCP/IP connection via JDBC driver. Executing queries directly saves additional overhead caused by creating new connections when the HTTP method is being used.

An important note to consider is the lack of ability to validate incoming triples to RDF Store, which is not related to Virtuoso by itself, but rather to all triple stores. One can validate triples via tools implementing the SHACL standard, but note that the primary focus of the standard is to verify that the object matches the given shape [28]. Based on comments on the reported issue[19] in the Virtuoso OpenSource Github project, there are no plans to add SHACL support for the open-source version. Hence it is up to the programmer to provide integrity constraints. Note that missing integrity constraints do not implicitly mean that the database engine by itself is not robust. It is mainly because the Semantic Web is based on the open-world assumption, whereas SQL is based on the closed-world assumption [29].

3.3 React

React, also known as *React.js* or *ReactJS*, is a Javascript library for building user interfaces for web applications. The main developer of React is Meta, although the community is very active. [30]

The main pillar of React are reusable components with their own state, logic, and render function. Every component must return an HTML element, Javascript primitive or React component. The whole application thus consists of an *n-ary* tree structure, where nodes are React components and leaves are JavaScript primitives or HTML elements. That structure is then captured in a *Virtual DOM.* [30]

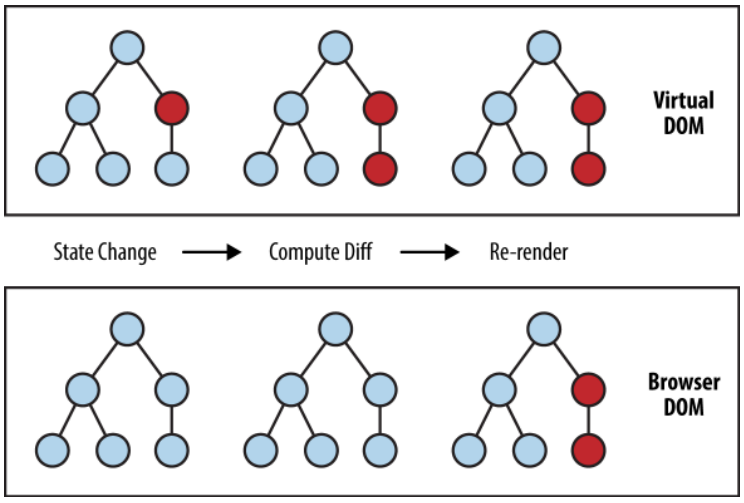


Figure 3.1: Propagation of state change from VDOM to DOM [31]

[19]https://github.com/openlink/virtuoso-opensource/issues/660

Figure 3.1 depicts the steps that occur right after the state change in a given component. When the state of one component changes, the *React's "diffing" algorithm* recalculates how those changes affect the related components in a given subtree. Based on those changes, the algorithm will first propagate changes to the VDOM and then modify the real DOM structure. This approach minimizes the amount of modifications and thus leads to better performance.

Comparision

It is worth noting, that for developing modern web applications, one can choose other libraries or either frameworks. According to the *Stack Overflow* survey [32] *React* is the most popular (40.14 %) library for building web application, followed by the *Angular* (22.96 %) and then by *Vue.js* (18.97 %). This statement is supported by total amount of downloads between 2015-2022 (see Figure 3.2).

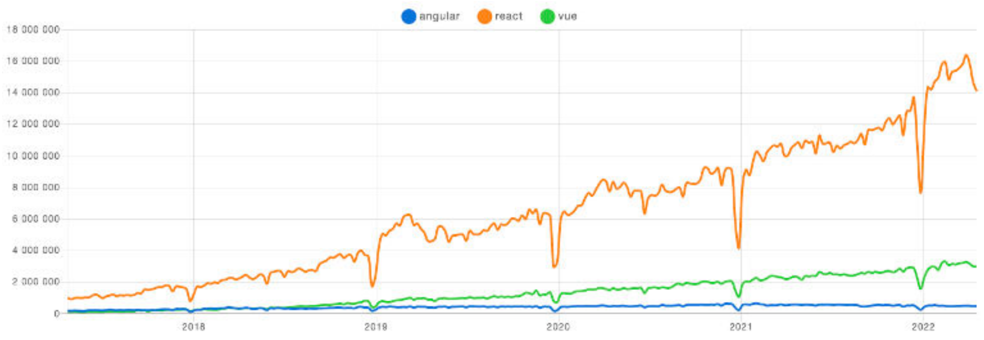


Figure 3.2: Comparison of popularity between the Angular, React and Vue [33]

Apart from the popularity perspective, we should address some major differences between these three. First *Angular* and *Vue.js* both provide a templating system on top of HTML [34, 35] whereas *React* does the templating in the JavaScript directly, with the JSX and uses one-way binding [30], whereas *Angular* and *Vue.js* supports two-way binding [34, 35]. *Angular* defines how the project structure should look [34], whereas *Vue.js* and *React* leave this decision up to the developer. In summary, *Angular* is suitable for enterprise-grade applications because of its steeper learning curve and modular system. In comparison between *React* and *Vue*, one might prefer to go with *Vue* because of no need to learn completely new syntax, whereas one might choose React with its more complex JSX that is more natural within the *JavaScript* environment. Based on the overall popularity, ecosystem and syntax, the *React* has been chosen as a core library for the web application.

3.4 NestJS

NestJS stands for a progressive Node.js framework for building efficient, reliable and scalable server-side applications written in TypeScript. [36]

The framework provides the out-of-the-box architecture for creating server-side applications like an API or microservices. For HTTP communication, the framework, by default uses Express as an HTTP adapter but can also be configured with *Fastify*. [36]

Every application based on NestJS follows modular architecture. Each module contains application domain-specific features. Having an organized project like this ensures you follow the *SOLID principles*. Every module is described by its module definition, which consists of four parts: [36]

- **controllers** - list of controllers implemented by this module,
- **providers** - list of services,
- **imports** - list of external modules used by services/controllers inside this module,
- **exports** - list of providers (services) that are allowed to be imported from other modules.

Because of the following structure, the framework can handle Dependency Injection and resolves potential circular dependencies. One can observe that certain parts of the module definition encourage the developer to use MVC architecture when creating an API.

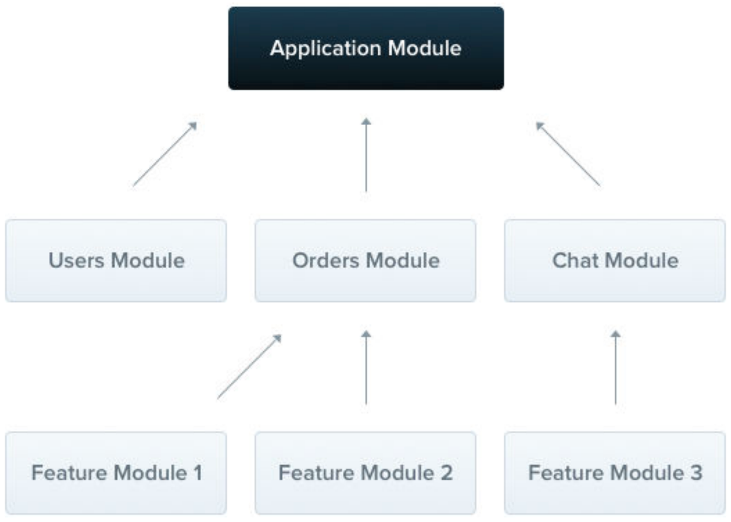


Figure 3.3: Preview of NestJS modules architecture [37]

NestJS provides a set of sub-packages and built-in utilities. The following list briefly introduces and describes the features that the NestJS comes with.

- **Validation/Transformation** pipes are built-in framework functions located in a core package called *@nestjs/common*. Validation pipes are compatible with *class-transformer* and *class-validator* packages. Those packages export decorator functions that are later applied to DTO or Entity properties. When the request comes in, the *ValidationPipe* and *TransformPipe* catch it and transform the request into an instance of a class that corresponds to a desired DTO or Entity. This means that the development is spared from doing manual validation in the controller or other types of services.

- **Caching** module is contained in a core *@nestjs/common* package and provides declarative configuration of cache storage without any direct manipulation. One can register *CacheModule* with an in-memory cache, file system or configure a caching module to be used with one of the supported databases like Redis/MongoDB/Couchbase or Hazelcast.

- **Config** module is one of the main modules providing a tiny wrapper around the *dotenv* package, which takes care of parsing appropriate *.env* files containing environmental variables for a given environment.

- **Testing** utilities are provided via the *@nestjs/testing* package, which exports a set of functions for easier creation of test suites with support for core framework features, like Dependency injection. The package also provides functions for mocking parts of the NestJS module dependencies.

- **Queue** module from *@nestjs/bull* is a wrapper around *bull*[20] that enables the developer to use Message Queues design pattern and thus split and scale application into multiple smaller parts.

Alternatives

Besides the *NestJS*, there are more frameworks that one can use. The most well known is the *Express*, which is very lightweight [38], supports middlewares and provides a routing mechanism. However, in contrast to other frameworks, it is low-level and thus requires more tooling for a larger application. The more advanced frameworks are *Koa* and *AdonisJS*. Both of them provide more declarative notation of common tasks imposed on creating the API [39, 40]. In addition to these functionalities, *NestJS* also provides the more abstract architecture that shades the programmer of the low-level parts, like manual parsing of the request body, validation or throwing error responses. The next major difference is that the only one that uses the Typescript decorators is well

[20]Redis-based queue for NodeJS

known in other programming languages. Based on the support of decorators for almost every provided functionality, MVC architecture, built-in support for request validations, automatic API and rich ecosystem, *NestJS* was chosen as the core technology.

3.5 Redis

Redis is an open-source, cross-platform, in-memory data structure store written in *C* language. The advantages include running atomic operations over data, periodical database dump to a disk or external storage, asynchronous replication and auto-reconnection. Community and Redis core has developed a set of client packages that enable the use of Redis from multiple programming languages. [41]

Because of the internal implementation of Redis as being key/value storage fully running in a memory, the database lookup with a known key is a fast operation. This is an ideal scenario for use cases like caching or queues.

3.6 Bull

Bull is a Node library that implements a fast and robust queue system based on Redis. The key concept of Bull is a *queue* and the base work unit is called a *job*. The queue can be created by simply creating an instance of a Bull class with the name of the queue as a constructor parameter. A queue instance can be responsible for up to 3 roles. The roles and appropriate responsibilities are summarized in the following list: [42]

- **Producer** responsibility is to add new jobs to one or more queues.
- **Consumer** or sometimes called worker receives jobs from one or more queues.
- **Listener** listens to selected events that are being emitted from the Bull mechanism. Those caught events can be used for logging or statistical reporting.

From an implementation perspective, giving a specific role to a queue is registering a callback function via exposed methods from the queue instance. Queue instances can thus be created multiple times. The queue instance with an already existing name and without registered events does not create additional entries in a Redis store [42]. However, it creates a new Redis connection.

When a job is being created and added to the queue, the job enters the lifecycle depicted in Figure 3.4.

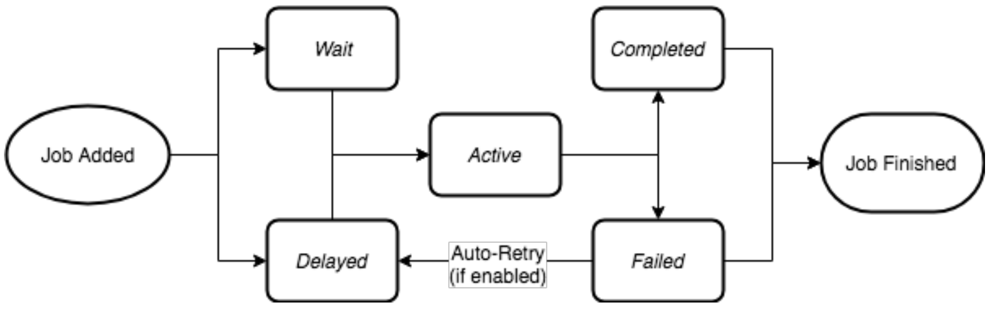


Figure 3.4: Bull Job lifecycle [42]

The given job can carry additional meta-information, which will influence its behaviour. One of those properties is prioritization, where jobs with higher prioritization get processed faster. Insertion of such a job is $O(n)$ [42], where n stands for a number of jobs currently waiting in the queue. The job can also be delayed, periodically run or even repeated in case of failure.

3.7 Monorepo

Large applications are often split into smaller pieces which are versioned inside separate GIT repositories to efficiently split responsibilities between different teams and thus improve the overall development speed. In the context of a web application, the common pattern is to split Frontend (UI - the web application) and Backend (API) into separate repositories.

Data transfer between the web application and the appropriate server is established via an HTTP connection. Related requests/responses should be well documented via appropriate technologies. The problem can occur when API introduces changes in the shape (interface) of the response data. This would cause a hard brake of the web application. It is also worth mentioning that those interfaces must be copy-pasted between projects which violate the DRY principle. The DRY principle is even more violated where multiple services use the same core parts like database connection.

One of the possibilities to solve these problems is to publish shared parts of the whole project inside installable packages, which will then be marked as dependencies in other projects. Having that flow delegates part of the problem to the package manager system like *Yarn* or the most popular NPM. This solution works till the point when test workflow comes in because the applications are still separate and not forced to use the newer package version used in production. The next concern includes E2E testing. Adding commits to the main branch can succeed in the committed project, but the introduced change can brake the dependent project.

The key concept of *Monorepo*, also known as *shared codebase* is having a single repository where applications (packages) are stored inside. Having a structured project like this results in having the following benefits: [43]

- **Code reusability** – Similar functionality or core features can be abstracted into shared libraries and directly included by projects.
- **Simplified dependency management** – In a multiple repository environment individual repositories often depend on same packages or shared modules. The bundle size in a monorepo can be thus reduced.
- **Large-scale refactoring** - Since developers have access to the entire project, refactoring individual modules can be tested and tried out in a single place.
- **Improved testing** - Given CI/CD has access to the whole repository and thus is able to run appropriate tests at one place.

Lerna

The *Lerna* is a library for managing JavaScript projects with multiple packages. Main functionality is provided by set of commands for bootstrapping and running commands across multiple repositories. [44]

Those commands can also serve as a tiny interface for more straightforward use. For instance, one can start all applications via a single command.

Chapter **4**

Design

This chapter describes the system's overall architecture, responsibilities of individual services and how they communicate together. The design emphasises the division of responsibilities according to requirements and the overall scalability of the system.

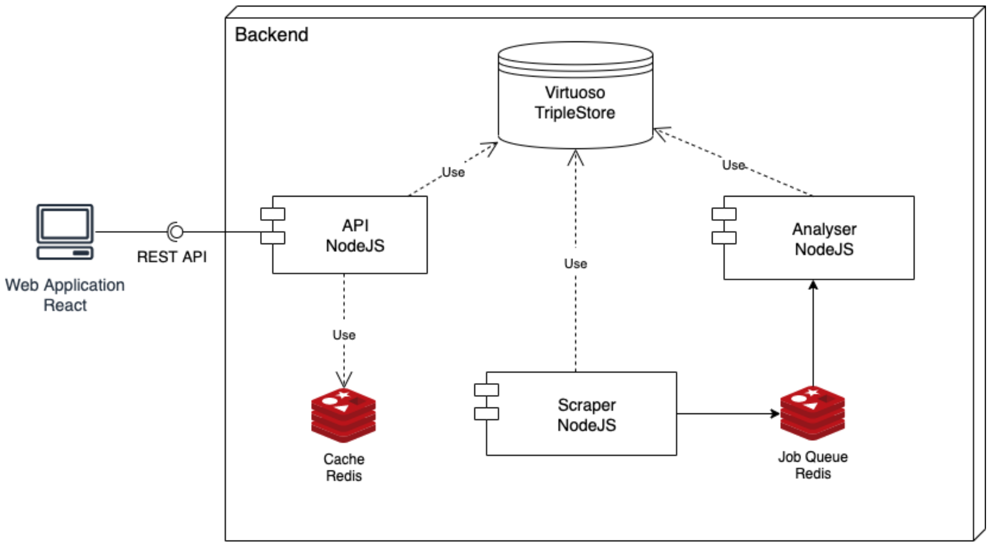


Figure 4.1: Architecture diagram of the project

Figure 4.1 depicts the overall system architecture. There are four applications - *Client (Web Application)*, *API*, *Scraper* and *Analyser*. The only visible part of the whole system is the API, which provides data for the visualisation in the *Web Application*. The API uses cache mechanism backboned by the Redis. Scrapped data from the real estate portals are sent from the *Scraper* application to the Redis queue, where those jobs are consumed in the *Analyser*. The *Analyser* then validates and saves data to the database.

4.1 Monorepo architecture

Because the whole project will be based on a single programming language (TypeScript), it is beneficial to keep the whole project inside a mono repository for simplified core maintenance and to ensure type safety across packages.

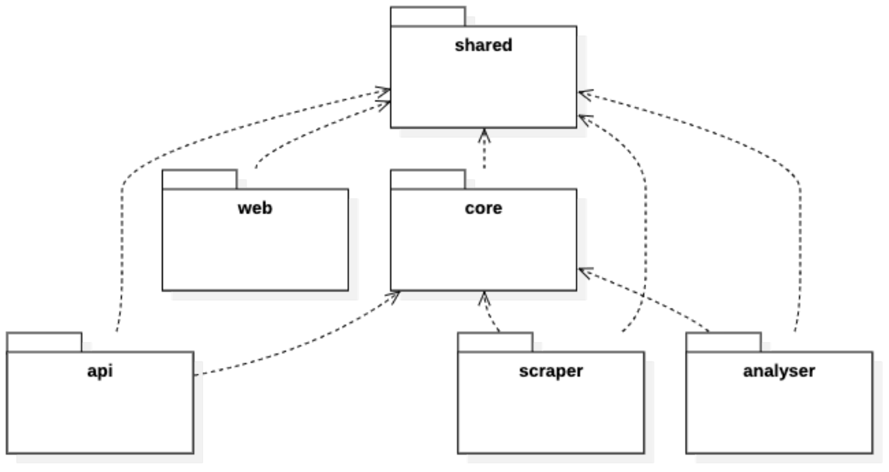


Figure 4.2: Diagram capturing packages and their dependencies

- **Shared (library)** package should contain only interfaces representing entities, server responses, types and predefined constants. Changing them influences other parts of the system. For instance, one of the interfaces can represent the real estate; changing it will influence system parts.

- **Core (library)** package should contain functionality purely for server services, like entities classes, database-related services or utility functions.

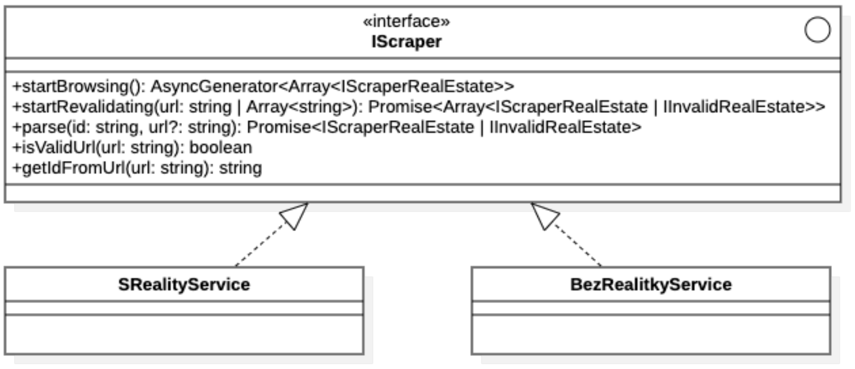
- **Scraper (application)** package is responsible for scraping and revalidating data from configured real estate portals. Those processed entities will be sent via a queue to the *analyser* application.

- **Analyser (application)** processes jobs from the queue and makes the appropriate changes to the database.

- **API (application)** package re-uses functionalities from the *core* package and provides REST API endpoints for the web application (contained in *web* package).
- **Web (application)** powered by *React* provides a user interface for clients. The package is dependent on *share* package, which contains interfaces that the application can expect from an API.

4.2 Data scraping architecture

Source data can come from various sources, which can change over time. Because of this, the scraping and storing tooling are separated into distinct packages. The *scraper* package contains the single orchestrator, which by the given configuration file loads appropriate sub-services representing the given portal to scrape data from. This would be done via a single interface that the concrete service then implements.



```
+startBrowsing(): AsyncGenerator<Array<IScraperRealEstate>>
+startRevalidating(url: string | Array<string>): Promise<Array<IScraperRealEstate | IInvalidRealEstate>>
+parse(id: string, url?: string): Promise<IScraperRealEstate | IInvalidRealEstate>
+isValidUrl(url: string): boolean
+getIdFromUrl(url: string): string
```

Figure 4.3: Diagram capturing scraper interface

The methods depicted in the Figure 4.3 are described as follows:

- **startBrowsing()** method browses over all adverts for the given platform.
- **startRevalidating(url)** method must be able to parse a given set of URLs and throw an error if some of them cannot be processed (for example, because they are not from the given portal).
- **parse(id, url?)** method scrapes the concrete advert; every mentioned portal in Section 1.1 uses an internal identifier for each advert. The service should be able to compose final URL just with it. The optional *url* part is provided for ability to track any changes to the original URL.

- **getIdFromUrl(url)** firstly validates if the given URL comes from the implemented portal and then parses the identificator of the resource.
- **isValidUrl(url)** validates if given URL belongs to the given portal. The purpose of this method is mainly for locating responsible service for given resource.

We described the main interface and purpose of each individual scraper service. Those services can be easily tested as no dependencies are needed and because they are just responsible for retrieving data. The implementation methods can vary based on the implementation of a given portal.

4.3 Data processing flow

The previous section described how the *scraper* package is structured and what each sub-module should look like. We did not cover how those tasks should be orchestrated. For running jobs, the package contains *core* and *cron* modules. The *core* service behaves as a facade and hides implementation details. One can give the *core* module a random URL and it finds out which service knows how to process it. The whole scraping/revalidating workflow will be started via a provided cron service, which the *scraper* should contain.

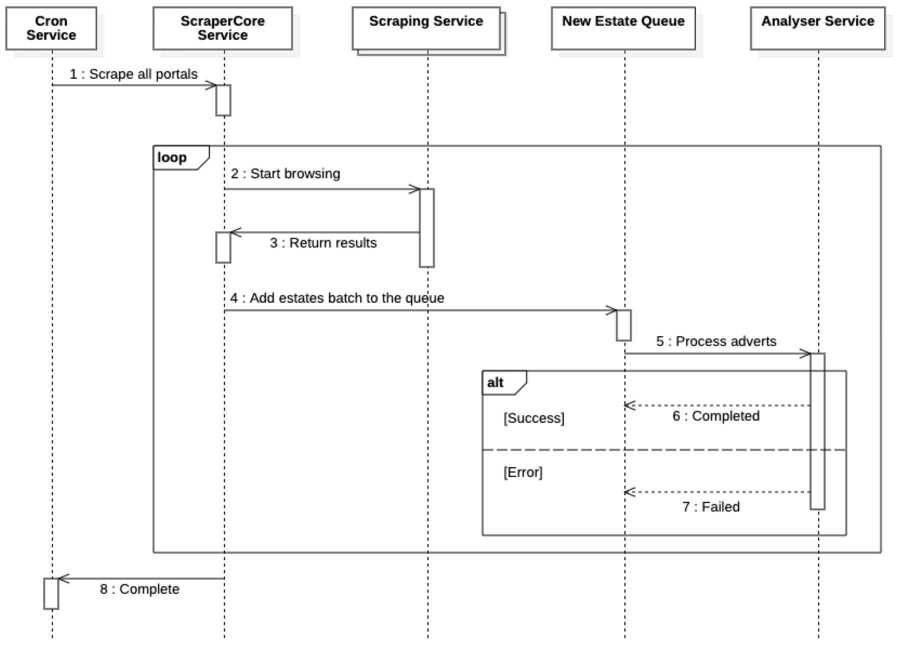


Figure 4.4: Communication between the *scraper* and *analyser*

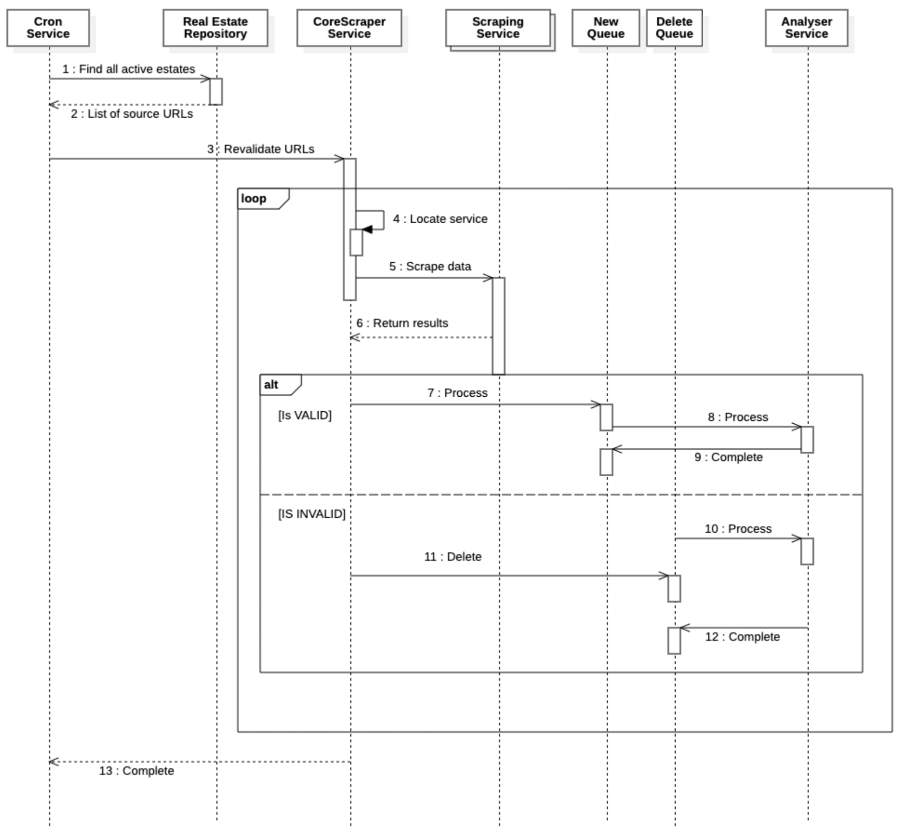


Figure 4.5: Sequence diagram capturing revalidating existing entities

Based on the analysis, not all data about specific real estate is contained in a straightforward parsable way. In other words, some required data can be contained in a free-text form, and to understand the context, the *Natural Language Processing* (NLP) methods shall be used. The key methods of NLP are the following: [45]

- **Tokenization** is a task of splitting a given sequence of characters into smaller pieces, called tokens. Tokens based on context are usually referred as words. Those tokens are also cleared from a certain character, such as punctuation.
- **Stemming** is a process of removing the end of the word (affixes) to return its root word. This technique does not use vocabulary and thus returns a word that does not exist.
- **Lemmatization** is similar to *Stemming*, but the result is a base of the word. The Lemmatization process uses vocabulary and morphological analysis of the word.

4.4 Data storage and retrieval

Figures 4.4 and 4.5 show that scraped entities go from the scraper module to a queue and they are then retrieved by processor in *analyser* package. When the job arrives, it will be removed. However, when a new estate queue retrieves a job, it can lead to a new entity creation or an entity update. In other cases a job will be ignored if there are no changes detected or the advert is a duplicate of another advert.

First of all, let us see how entities are represented; note the interfaces definition. Using interfaces prior to classes is easier for integration with web applications where serialisation is present due to the data transfer over the network.

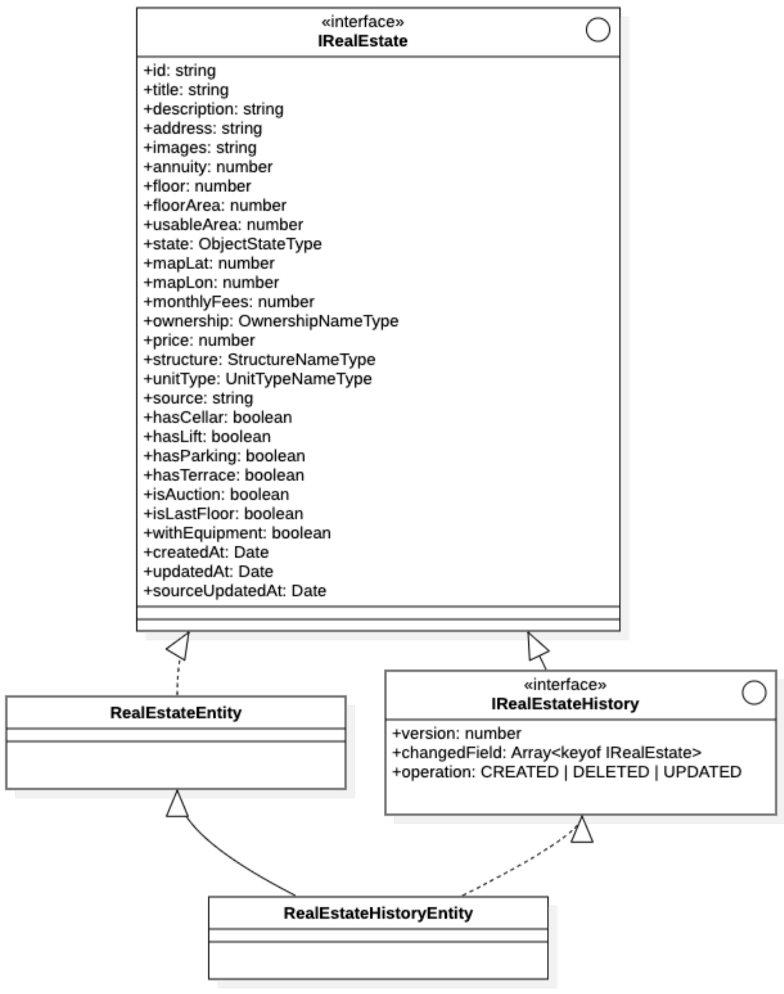


Figure 4.6: Preview of two core entities of the project

Note that we are not using a relational database, and thus, the data will not be normalised across multiple tables as is a known pattern for relational databases. In fact, the corresponding data are triples, as shown in Section 1.2.1. The responsibility for storing those data in a given format is up to the repository service; thus, one can change the storage to another triple store or even to a relational database, and only the underlying repository must change.

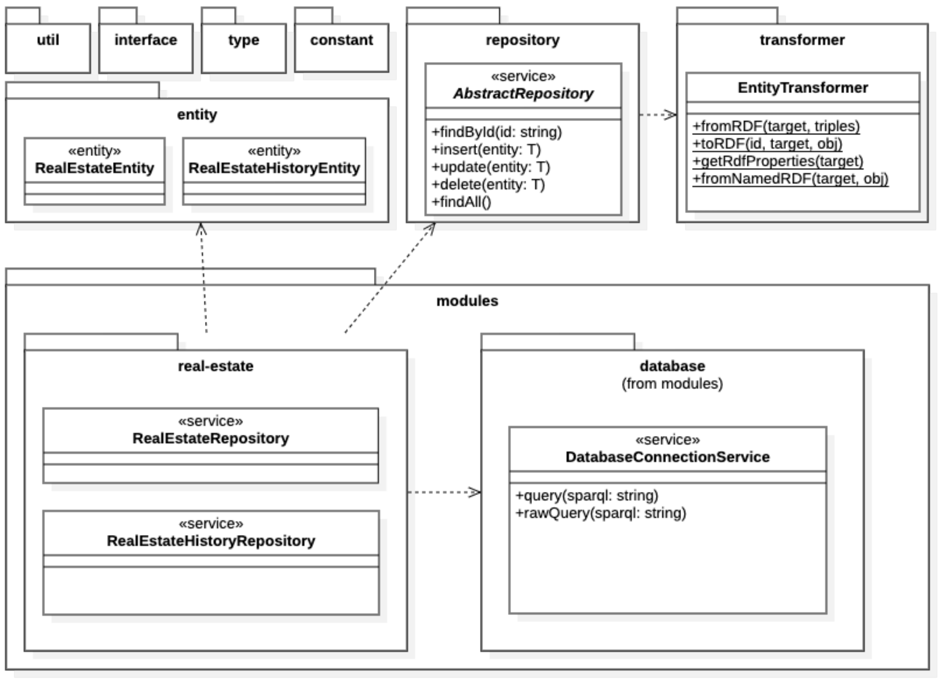


Figure 4.7: Preview of components grouping in a core package.

Modules can be imported as dependencies and configured independently. Because of the lack of native integrity constraints in triple stores like Virtuoso, the integrity constraints must be validated inside a responsible repository. It is the repository because it is the last part that directly sends queries to a database.

4.5 Web application

The web application is the only part of the system that the client will directly access. The design process was done in the form of wireframes. Based on the requirements defined in Chapter 2 (Requirements), the web application will contain the following pages:

- **Home** - introduce the project,
- **Search** provides search form with filters,
- **Listing of adverts** contains a listing of real estate with pagination and an interactive map with markers,
- **Detail of advert** depicts all properties of the real estate with information about its surrounding.

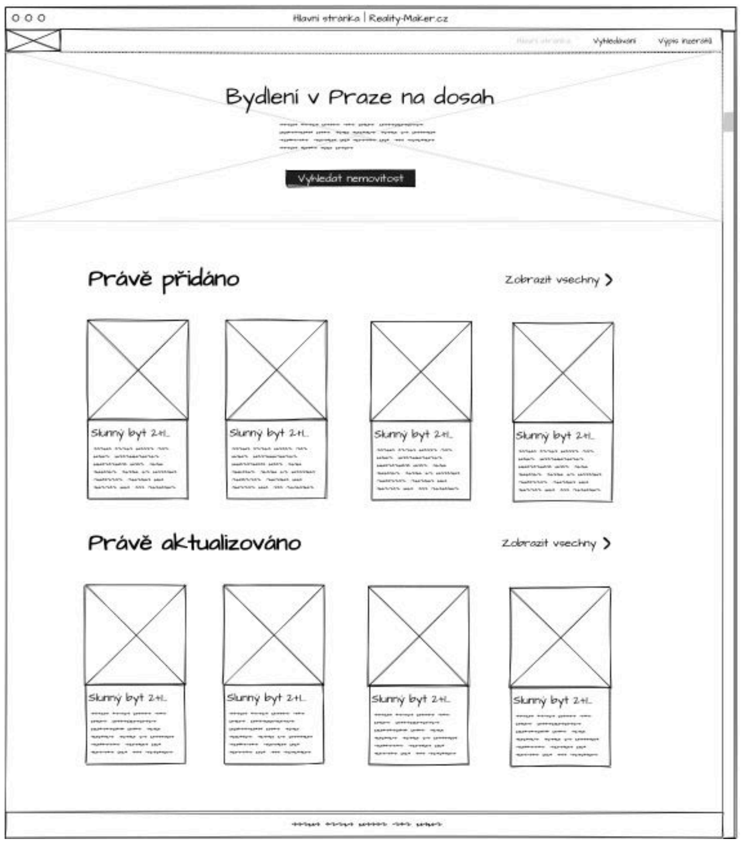
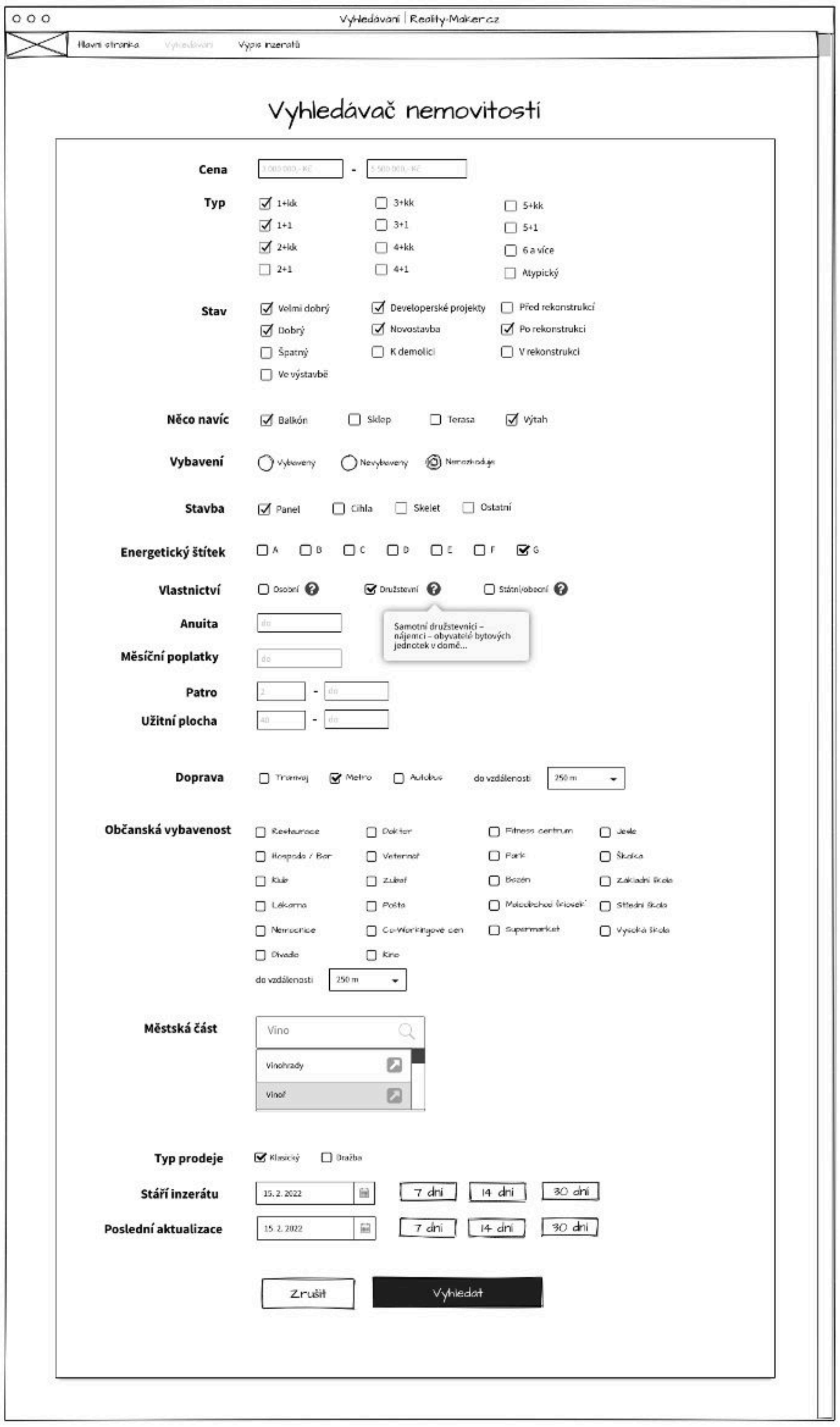


Figure 4.8: Wireframe for the homepage of the web application

The homepage should welcome users and give them a brief description of the project. The page will then lead the user to visit a search page via the action button.



Vyhledávání | Reality-Maker.cz

Hlavní stránka · Vyhledávání · Výpis inzerátů

Vyhledávač nemovitostí

Cena	1 000 000,- Kč - 5 500 000,- Kč
Typ	[✓ 1+kk] [✓ 1+1] [✓ 2+kk] 2+1 · 3+kk 3+1 4+kk 4+1 · 5+kk 5+1 6 a více Atypický
Stav	[✓ Velmi dobrý] [✓ Dobrý] Špatný Ve výstavbě · [✓ Developerské projekty] [✓ Novostavba] K demolici · Před rekonstrukcí [✓ Po rekonstrukci] V rekonstrukci
Něco navíc	[✓ Balkón] Sklep Terasa [✓ Výtah]
Vybavení	Vybavený Nevybavený [✓ Nerozhoduje]
Stavba	[✓ Panel] Cihla Skelet Ostatní
Energetický štítek	A B C D E F [✓ G]
Vlastnictví	Osobní ? [✓ Družstevní] ? Státní/obecní ?
Anuita	do
Měsíční poplatky	do
Patro	2 - do
Užitní plocha	40 - do
Doprava	Tramvaj [✓ Metro] Autobus · do vzdálenosti 250 m
Občanská vybavenost	Restaurace, Doktor, Fitness centrum, Jesle, Hospoda / Bar, Veterinář, Park, Školka, Klub, Zubař, Bazén, Základní škola, Lékárna, Pošta, Maloobchod řetězec, Střední škola, Nemocnice, Co-Workingové cen, Supermarket, Vysoká škola, Divadlo, Kino · do vzdálenosti 250 m
Městská část	Vino — Vinohrady, Vinoř
Typ prodeje	[✓ Klasický] Dražba
Stáří inzerátu	15. 2. 2022 · 7 dní · 14 dní · 30 dní
Poslední aktualizace	15. 2. 2022 · 7 dní · 14 dní · 30 dní

Samotní družstevníci – nájemci – obyvatelé bytových jednotek v domě...

Zrušit · Vyhledat

Figure 4.9: Wireframe for the search page of the web application

Figure 4.10 depicts what the user can expect after submitting the search form mentioned in Figure 4.9.

The left side of the screen will be filled with an interactive map where each real estate will be depicted as a marker. After the data loads, the map will be justified on the centre and reasonably zoomed-in regarding distances between markers. The user will then be able to scroll, zoom and drag the map in any direction. Clicking on the marker will result in the display of a detail page.

The right side of the page will contain a list of real estate matching the user's filters. The list starts with a header and gives the user the ability to return to the search page while preserving[21] filter parameters. The item then includes price, title, location, brief description and image carousel with the ability to scroll to the left and right. Images can vary in height and width, so the carousel must accommodate that. Finally, the list contains pagination with a customizable page size. Pagination changes should also be preserved and therefore be used for link sharing.

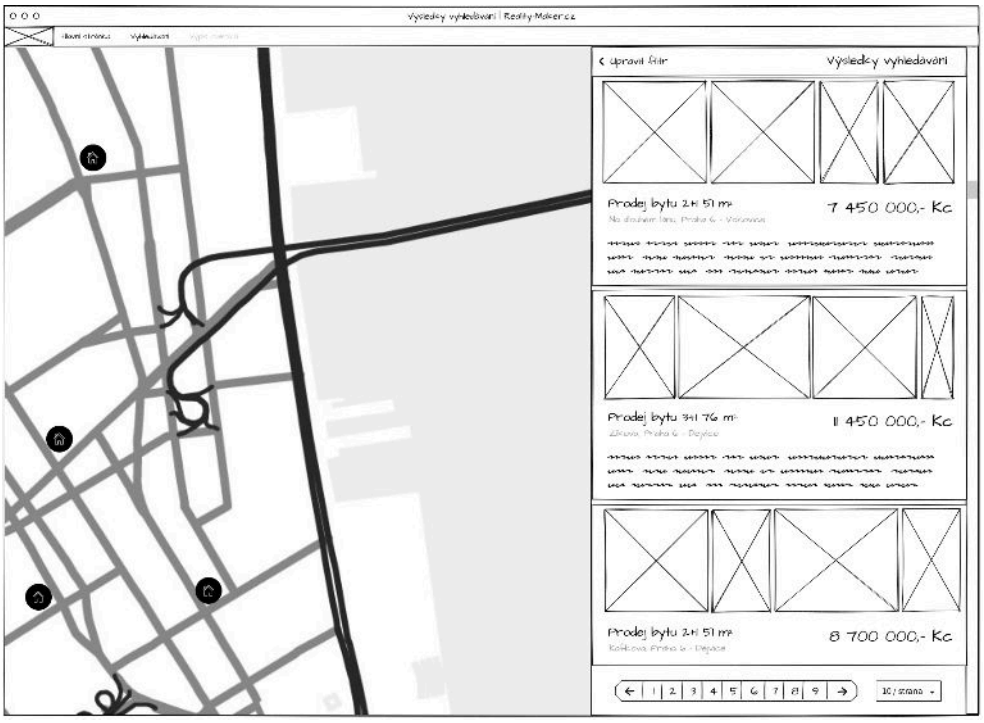


Figure 4.10: Wireframe for displaying search results

Figure 4.11 and 4.12 introduces the detail page. The beginning of the page is in a similar style as a list view depicted in Figure 4.10. The reason for that is consistency. From a detail page, the user can go back to the search results

[21]storing as query parameter

via the link button in the header or via the browser's go back button. After the load process, the map should align itself to a selected marker.

The page provides a button to show a modal window with all changes which have occurred. Figure 4.12 shows how civic amenities and the public transport are displayed. They should also be clickable, and related data will be shown in a modal dialog.

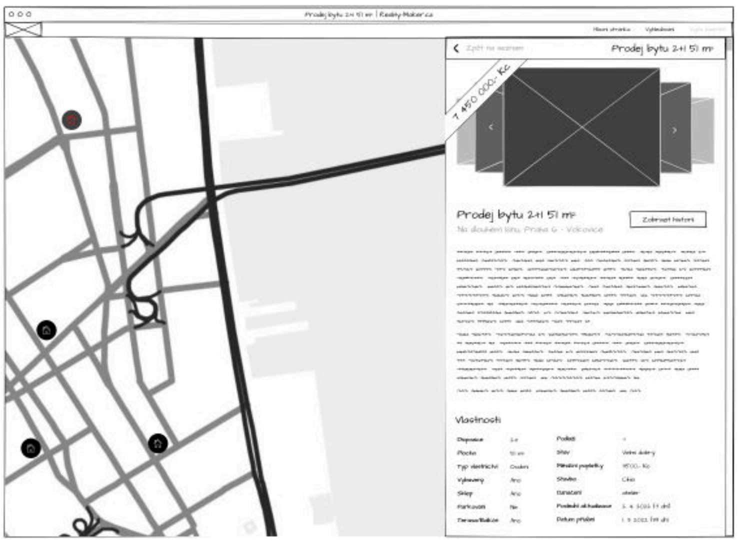


Figure 4.11: Wireframe for displaying search results (top)

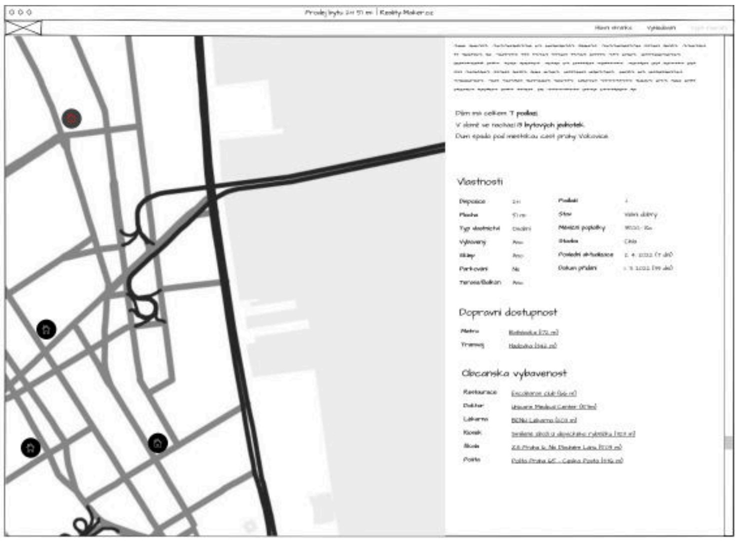


Figure 4.12: Wireframe for displaying search results (bottom)

4.5.1 API Design

The *api* package should serve as an API for the web application. The API will be designed in REST [46] architectural style. To increase interoperability, the API will follow *json:api* specification. The API will not take full advantage of the specification; hence, it does not use relation conventionally. Although, the API will not violate any of the specification rules.

The API will reuse exported modules from the core package and build its own functionality on top of them. In general, the API will contain modules oriented on resources, where each module consists mainly of controllers and services, where the controller receives the request and, based on internal knowledge, delegates the processing to the appropriate service and finally responds with a JSON[22] data format.

Wireframes defined in Section 4.5 implicitly define the requirements for an API. Those requirements are the following:

- list of real estate based on filters coming from the search form,
- list of a real estate nearby,
- details of the real estate by a unique identifier,
- social facilities and public transport (collectively referred to as POIs) around given real estate,
- full detail of given POI,
- full history of a given real estate with information about changed fields in every revision.

Figure 4.13 depicts API endpoints exposed to a web application based on requirements specified in an enumerated list above.

[22]concretely in *application/vnd.api+json* format to match *json:api* specification

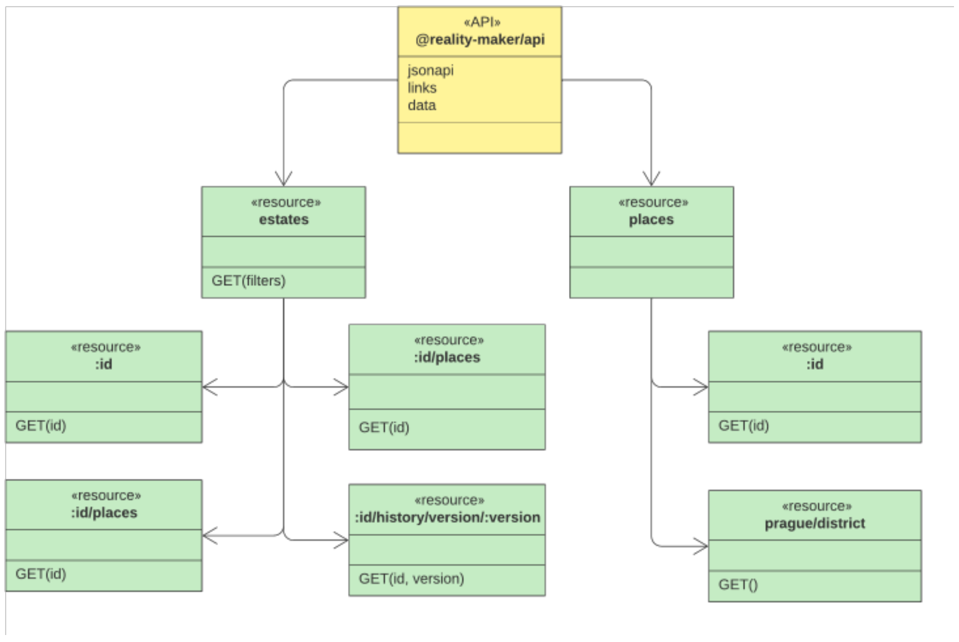


Figure 4.13: Preview of the API endpoints exposed to the web application

Note that the *filters* query parameter used on *estates* endpoint must be complex enough to cover the needs of the search form. Filters can include fields related to real estate interface or even virtual ones, like filtering estates falling under a given city district. The ideal scenario would be to use an isomorphic serialization on API and the web app.

Following the *place* detail, endpoint does not contain given fields. That is because the interface cannot be fully determined, as the source of places (POIs) will come from OSM or another source with a flexible data format. Only some common properties are known. In other words, we can be sure that a tram station and a supermarket have a name, but we also know that they do not have all properties in common, like opening hours.

CHAPTER 5

Implementation

This chapter covers the implementation of the architecture described in the Chapter 4 (Design). The chapter describes the monorepo directory structure, followed by the implementation of *shared* and *core* packages because other packages depend on them. Lastly, the chapter describes the implementation of each application (*Scraper*, *Analyser*, *Web* and *API*).

5.1 Project setup

For managing the monorepo architecture, the *Lerna* package, described in Section 3.7, has been used. This is mainly because of its ability to track dependencies between packages and thus is able to build a single application without building the unnecessary ones, the same applies to application bootstrapping. Figure 5.1 depicts the overall project directory structure.

```
│ lerna.json
├─ package.json
├─ libs ........................................ non-runnable libraries
│   ├─ shared. ................................. used by all applications
│   └─ core. ................................ used by server applications
└─ apps ........................................... runnable applications
    ├─ scraper
    ├─ analyser
    ├─ api
    ├─ web
    └─ importer ..................... set of CLI functions for import/export
```

Figure 5.1: Top level overview of project's directory structure

The *lerna.json* file defines the name of the package manager (npm or yarn), versioning policies and locations where sub-packages can be found; in our case, the directories are *libs* and *apps*. Lerna needs to know where the packages lie due to dependency management, symlinking packages and, last but not least, for running commands across all projects from a single place.

The *package.json* then contains meta information and mainly a list of dependencies and scripts. Because of Lerna, scripts can be defined as runnable across all packages. It helps bootstrap and starts all applications at once.

In addition to the files mentioned above, the root directory also contains configuration files for code style (*Prettier*, *ESLint*). Those files are not described because they have no impact on code functionality but instead on formatting.

Finally, the root directory contains various *Dockerfiles* (for each project), *Docker Compose* (to build and run all containers) and *PM2* configuration files.

5.2 Core packages

The *shared* package provides interfaces and shared constants. The package is used in all other packages. Figure 5.2 depicts directory structure.

```
│ package.json
├─tsconfig.json
└─src
  ├─index.ts
  ├─constants
  │ ├─index.ts
  │ └─RealEstate.constants.ts
  └─interfaces
    ├─index.ts
    ├─IPlace.ts
    ├─IRealEstate.ts
    ├─IRealEstateHistory.ts
    ├─ITransport.ts
    ├─IPlace.ts
    ├─ITransport.ts
    ├─IHistory.ts
    └─IWikida.ts
```

Figure 5.2: Directory listing of *shared* package

Note the *index.ts* file in each subdirectory. Their purpose is to choose those interfaces/constants exposed to the upper layer. Because every directory follows this rule, the *src/index.ts* file exports all from underlying sub-directories. In other words, every directory[23] takes the responsibility for the outputs.

[23]or module, depending on context

In contrast with the *shared* package, the *core* package provides interfaces, constants, utility functions and mainly database related services which are used in other server-side packages. The *core* package is based on NestJS framework described in Section 3.4. Following Figure 5.3 depicts the overall[24] directory and file structure.

```
│ entity
│  ├─ Abstract.entity.ts
│  ├─ RealEstate.entity.ts
│  └─ RealEstateHistory.entity.ts
├─ decorator
│  └─ SemanticProperty.decorator.ts
├─ transformer
│  └─ Entity.transformer.ts ........... Triples to classes and vice versa
├─ repository
│  └─ Abstract.repository.ts ................ provides CRUD operations
├─ constants
│  └─ namespaces.ts ............. the graphs and vocabularies namespaces
└─ modules
   ├─ database
   └─ real-estate
```

Figure 5.3: Directory listing of *core* package

Services or functions that do not depend on NestJS functionalities or NestJS modules are located outside the *modules* directory.

5.2.1 Database connection

The database connection module located in the *core* package is responsible for providing a database connection to the *Virtuoso* database. Besides that service exposes two methods for sending queries. The most important is the *query()* method, which takes a SPARQL query as a parameter and executes it directly on the database and based on query syntax analysis, the method decides how the response should be formatted. For *SELECT/CONSTRUCT* query types, the array of triples is returned. For *ASK* queries the *boolean* value is returned, otherwise *null*.

We mentioned in Section 3.2 that when an application does frequent queries, it may be worthwhile to create a direct database connection, which will be reused across multiple queries. Since there is no native *Virtuoso* driver for *NodeJS*, the only option is to use *Java* bindings, which enables communication between a JDBC driver and *NodeJS* application. The *@naxmefy/jdbc*[25]

[24] the not important files are omitted - like style configs or even some *index.ts* files
[25] `https://github.com/naxmefy/node-jdbc`

```
│ DatabaseConnection.module.ts
├── DatabaseConnection.service.ts
├── DatabaseConnection.child.ts
├── IDatabaseConnection.config.ts
├── types.ts
└── utils
    ├── initJVM.ts
    └── VirtuosoResult.parser.ts
```

Figure 5.4: Directory listing of *core* package

package provides a support for a communication with JDBC driver, which is appropriate because *Virtuoso* provides a JDBC driver and thus they can be used together. The only disadvantage of this approach is that the process needs to start JVM and whenever it crashes, it crashes the whole application.

To prevent the crash of the whole application, the *DatabaseConnectionService* manages a pool of child processes (*DatabaseConnectionChild*) which are responsible for the queries execution. Selection of the child process is made by *Round Robin load balancing* mechanism [47]. Delegation of the execution is done in the form of sending messages through the IPC channel, with a unified interface defined in *types.ts* file. Once the query inside the message is being resolved by the child process, it sends a new message with the response back to the service. In case of a crash of the JVM, for instance, due to the broken connection with the database or segmentation fault, the child will be terminated and deleted from the pool. Because the child processes are detached from the parent, they do not crash the application, but only themselves.

The *DatabaseConnectionModule* is a *NestJS* module that encapsulates the services within a module and exports only service for the database connection. Besides that, the module provides static methods for configuration so that the dependent package can provide its configuration options (server, username, password, pool size and so forth). The configuration option interface is contained in *IDatabaseConnection.config.ts* file. Once the module has been initialized, it can be used across the application via the dependency injection mechanism.

5.2.2 Entity representation

To represent entity in a *triple-store*, we first need to define the vocabulary with terms that we will use later on. The definition of the project's vocabulary can be seen in Figure C.2 and the following usage in Figure C.3. The appropriate HTML documentation for the vocabulary can be generated via *ontospy*[26] package. The generating command and appropriate documentation are included within the project in the *README.md* file in the root directory.

To understand how SPARQL queries (in repositories services) are generated and how the integrity validation is achieved, we need first to look at how a given entity class is represented.

```
@Class(RealityMakerVocab.RealEstate)
@IriPrefix(RealityMakerEntity.RealEstate)
class RealEstateEntity extends AbstractEntity implements IRealEstate
{
  @IsString()
  @Type(() => String)
  @Property(ns.dcterms.identifier)
  id: string

  @IsString()
  @Property(ns.dcterms.title, {
    language: "cs",
  })
  title: string

  @IsNumber()
  @Min(200_000)
  @Transform(compose(parseFloat, get('value')))
  @Property(ns.RealityMakerVocab.price, {
    datatype: ns.xsd.nonNegativeInteger,
  })
  price: number

  @IsString()
  @Property(ns.schema.address)
  address: string

  @IsDate()
  @Type(() => Date)
  @Property(ns.dcterms.created, {
    datatype: ns.xsd.dateTime,
  })
  createdAt: Date
  ...
}
```

Listing 5.1: Entity class representation in a code

The code depicted in Listing 5.1 shows how one can achieve integrity constraints and semantic enrichment with decorators. First of all, the *@IsString*, *@IsNumber* and *@IsDate* decorators came from the *class-validator* package. Besides the decorator functions, the package contains a set of functions for transformations. One of those functions is *plainToClass*, which takes two parameters; a plain object and a reference to the desired class constructor. The function then resolves to which metadata are associated with given fields and does the transformation. If some properties do not match the defined validation schema, the function throws an error.

[26] https://github.com/lambdamusic/Ontospy

Next, the *@IriPrefix*, *@Class* and *@Property* are custom decorators implemented inside the package. Their purpose is to give the class with its properties a semantic meaning.

The **@IriPrefix** decorator function is the simplest one. It simply sets the IRI prefix for the subject. The way the query is built is implemented in the *AbstractEntity*. The implementation can be overridden in a subclass, in our case in *RealEstateHistoryEntity*. The default behaviour is to take the value of *id* property and append it after the prefix.

The **@Class** decorator function tells which RDF class the object is an instance of. For instance, *RealityMakerVocab.RealEstate* is a named node for *http://reality-maker.cz/vocabulary/RealEstatate*, then assume that IRI for our entity is *http://reality-maker.cz/entity/c032b866*. Based on the usage of the *@Class* decorator, the entity transformer will produce the following output.

```
PREFIX realm: <http://reality-maker.cz/vocabulary/>
PREFIX entity: <http://reality-maker.cz/entity/>

entity:c032b866 rdf:type realm:RealEstate .
```

Listing 5.2: Transformation to RDF with *@Class* decorator

Finally, the **@Property** decorator defines the corresponding IRI for a given class property. Besides it, one can also define the concrete datatype and language tag. If the data type is not specified, it will be derived based on the data type of the current value. The responsible transformer knows how to handle the conversion. To describe the full transformation, let us consider the following plain object.

```
{
  id: "c032b866",
  title: "Prodej bytu 3+1",
  price: 7500000,
  address: "Na Klimentce, Praha 6 - Dejvice",
  createdAt: "2022-03-27T20:00:40.507Z"
}
```

Listing 5.3: Partial *RealEstateEntity* as a *JavaScript* object

The plain object will be then via functions from *class-transformer* package transformed to a class instance, which then will be via *EntityTransformer* converted to RDF in *Turtle* serialization.

```
PREFIX realm: <http://reality-maker.cz/vocabulary/>
PREFIX entity: <http://reality-maker.cz/entity/>
PREFIX schema: <http://schema.org/>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>

```

```
entity:c032b866 rdf:type realm:RealEstate ;
  dcterms:identifier "c032b866" ;
  dcterms:title "Prodej bytu 3+1"@cs ;
  realm:price "7500000"^^xsd:nonNegativeInteger ;
  schema:address "Na Klimentce, Praha 6 - Dejvice" ;
  dcterms:created "2022-03-27T20:00:40.507Z"^^xsd:dateTime .
```

Listing 5.4: Conversion from a plain object to RDF

Note, that the conversion from the plain objects to one of RDF's is only done before sending the SPARQL query. Meanwhile they are represented as class instances which corresponds to interfaces defined in *RDF/JS: Data model specification*[27].

5.2.3 SPARQL queries

We have just shown how the convertor process works, but the SPARQL queries must be used for actual modifications to the database. That is achieved via *AbstractRepository* class, which provides base CRUD operations. Besides that, the repository implements the *Observer pattern* thus other services can register for listening on changes. This ability is then used for capturing the history of the advert and is realized by *RealEstateHistorySubscriber* class inside the *core* package.

Because of overall complexity and size of the *AbstractRepository*, only the usage will be described. The following Figure (5.5) shows the abbreviated version of *RealEstateRepository*[28].

```
class RealEstateRepository extends AbstractRepository<RealEstateEntity> {
  static get GRAPH_NAME(): NamedNode<string> {
    return rdf.namedNode('http://reality-maker.cz/graph/sell')
  }

  getEntityClass() {
    return RealEstateEntity
  }
}
```

Listing 5.5: Minimal database repository class representation

Note, that we only need to implement two methods to create a complete CRUD functionality over the entity. That is because the meta information needed for conversion to RDF are extracted from the appropriate entity class because of their meta decorators.

Because we extend *AbstractRepository* we inherit the CRUD functionality. One can specify which fields want to select, offset, limit, order and mentioned property filters. These filters will be translated to a SPARQL query in a base

[27]http://rdf.js.org/data-model-spec/

[28]*RealEstateHistoryRepository* is implemented in similar way

repository class. The following examples show how the filters are propagated to the final SPARQL query.

```
repository.findAll({
  select: ['id', 'price'],
  where: [{
      key: 'price',
      value: 200000,
      operator: '>',
  }, {
      key: 'price',
      value: 800000,
      operator: '<',
  }, {
    key: 'energyLevel',
    operator: 'IN',
    value: ['A', 'B'],
  }, {
    key: 'hasLift',
    operator: '=',
    value: true,
  }],
  order: [{
    key: 'price',
    desc: true
  }],
  offset: 10,
  limit: 20
})
```

```
PREFIX dcterms:
  <http://purl.org/dc/terms/>
PREFIX realm:
  <http://reality-maker.cz/vocabulary/>
PREFIX entity:
  <http://reality-maker.cz/entity/>
PREFIX schema:
  <http://schema.org/>
PREFIX xsd:
  <http://www.w3.org/2001/XMLSchema#>

SELECT DISTINCT ?id ?price
FROM
  <http://reality-maker.cz/graph/sell>
WHERE {
 ?entity a realm:RealEstate .
 ?entity dcterms:identifier ?id .
 ?entity realm:energyLevel ?energyLevel .
 ?entity realm:price ?price .
 ?entity realm:hasLift ?hasLift .
 FILTER (?price >
  "200000"^^xsd:nonNegativeInteger) .
 FILTER (?price <
  "800000"^^xsd:nonNegativeInteger) .
 FILTER (?energyLevel IN ("A", "B")) .
 FILTER (?hasLift = "true"^^xsd:boolean)
}
ORDER BY desc(?price)
LIMIT 20 OFFSET 10
```

Listing 5.6: Preview of conversion from code to SPARQL

Although the Listing 5.6 depicts the core search options, the underlying mechanism provides even further search capabilities. For instance: selecting data from multiple graphs, passing custom filter expressions or casting data types.

5.3 Scraper

The *scraper* package's responsibility is to scrape data from various real estate portals and send those results to the Redis queue, where are further processed by the *Analyser*. The *scraper* package supports scraping data from two real estate portals; *Bezrealitky* and *SReality*. Scraping from *Bezrealitky* is based on downloading plain HTML files which are then parsed via the *node-html-parser* package to create DOM, which is then processed with the help of CSS selectors. On the other hand, the scraping of *sreality.cz* is done via sending requests to a public (but not documented) API. Besides the *scraping* process, the package also contains a *cron* module for periodically starting the scraping of new adverts and revalidation of existing adverts.

The directory structure is similar to other NestJS applications in this project and is described in Figure 5.5.

```
main.ts.....................................application's entry point
├── bootstrap.ts...........................initialize NestJS application
├── app.module.ts
├── interface
│   └── IScraper.ts.................defines required interface for scrapers
├── utils...................................shared utilities for scraping
└── modules
    ├── core.................delegates the work to concrete scraper service
    │   ├── core.module.ts
    │   └── core.service.ts
    ├── cron...........periodically triggers scraping and data revalidating
    ├── sreality
    │   ├── sreality.module.ts
    │   ├── client
    │   │   ├── detail.processor.ts
    │   │   ├── list.processor.ts
    │   │   └── api.transformer.ts
    │   └── interfaces
    │       └── ISrealityApiResponse.ts
    └── bezrealitky
        ├── bezrealitky.module.ts
        ├── bezrealitky.service.ts
        └── client
            ├── constants.ts
            ├── detail.processor.ts
            └── list.processor.ts
```

Figure 5.5: Directory listing of *scraper* package

When the application starts it first initializes the base *app.module.ts*, which imports and sets up the the database connection (via exposed database module from the *core* package), *cron* module and internal *core* module.

The internal *core* module encapsulates the behaviour of all scraper services via a single interface depicted in a Figure 4.3. Once the module has been imported from the *core* module, it loads its related modules (*sreality*, *bezrealitky*), based on configuration in the *.env* file. Note that based on the configuration file, only those explicitly mentioned services will be loaded, which implies that in the cloud environment, the application can be scaled for each scraper service independently.

When one wants to parse a single URL, the receiver (*CoreService*) selects appropriate service by calling *isValidUrl* method on each scraper's services until one responds with *true* value, otherwise an error is thrown. This pattern is well known in OOP and leads to better code readibility, because the information does not have to be hardcoded inside the *CoreService*. Having large *switch*/*if* statements in a code violates the OCP. The following figures describe how data are retrieved from scrapers.

```
// scrapes all adverts from all portals
async startAll() {
  await Promise.all(
    this.services.map(async (scraper: IScraper) => {
      // asynchronously receive chunk of parsed adverts
      for await (const chunk of scraper.startBrowsing()) {
        // send those adverts to a queue
        await this.addToQueues(chunk)
      }
    })
  )
}
```

Listing 5.7: *startAll* method in *CoreService* (*scraper* module)

The previous Listing shows what the scraper's services are called. The following Listing shows what the concrete *SrealityService* scraper looks like.

```
async *startBrowsing() {
  const pageIterator = browsePagesFactory(CHUNK_SIZE)

  // receive listing of estates from SReality private API
  for await (const [estatesData, loadedItems, totalCount] of pageIterator()) {
    // for every received id, get the full detail
    const parsedEstates = await Promise.all(
      estatesData.map((estate) => this.parse(estate.hash_id))
    )
    // pause the execution by returning valid results
    yield parsedEstates.filter(isEntityToAdd)
  }
}
```

Listing 5.8: *startBrowsing* method in *SrealityService* (*scraper* module)

We just showed the communication flow between the *CoreService* and *SRealityService*. Now we will describe in more detail the functions that were used inside a *startBrowsing* method.

The **browsePagesFactory** function creates a generator that, on every iteration, returns a chunk of data, in JSON format, received from SReality API, generator end when the last page is processed.

On the other hand, the **parse** method does the transformation from a JSON data to a valid *IRealEstate* or *IInvalidRealEstate* interface. The method is internally implemented within two phases:

1. **Fetching** raw data from *Sreality* API with retry mechanism.

2. **Extracting** data to match the *IRealEstate* interface.

The extraction process starts with iterating over the properties of the received object. Each property key is normalized[29] to improve matching once the API that we have no control over changes. Every field is then checked, and if there is an implemented parsing mechanism for it, it is being processed, otherwise skipped. Once all properties have been processed, the parse will lookup up to the text fields (like description or price notes) and extract desired properties via NLP methods (normalization, tokenization and stemming). The property description is then parsed as follows:

1. remove HTML tags and language abbreviations,

2. split full description into sentences (via regex expression),

3. normalize, tokenize (with stopwords removal) and stem each sentence,

4. lookup if some stemmed words match given pattern (price, monthly fees, layout, structure), if so, update the parsed result.

The NLP methods are used from *node-nlp*[30] package which has a partial support for the Czech language. The reason why stemming is used over lemmatization has two reasons: the library does not support lemmatization but only stemming. The second is that it does not matter as long as we just parse a text without further sentence analysis.

Once the data are being processed, they are sent to the queue via *addToQueues* method. The way how they are processed will be described in the following section.

A similar process applies to a *bezrealitky.cz*, the only difference is that the data about adverts are not directly taken from an object (after JSON conversion) but the DOM via CSS selectors. The way they are then processed is the same.

[29] via *webalize* package

[30] `https://github.com/axa-group/nlp.js`

5.4 Analyser

The *analyser* package's responsibility is to process jobs from the Redis queue, created by scrapers.

The directory structure is similar to other NestJS applications in this project and is described in Figure 5.6.

```
main.ts........................................application's entry point
bootstrap.ts..............................initialize NestJS application
app.module.ts
utils
└── isDuplicate.ts
modules
└── processor
    ├── processor.module.ts
    ├── RemoveEstate.processor.ts
    └── NewEstate.processor.ts
```

Figure 5.6: Directory listing of *analyser* package

On the application bootstrap, the process first registers two queues, one for retrieving valid real estate and one for deleting. The Listing 5.9 depicts how adverts for deletion are consumed from the queue.

```
@Injectable()
@Processor(Core.Constant.REAL_ESTATE_REMOVE_QUEUE)
class RemoveEstateQueueProcessor {
  constructor(
    @Inject(Core.Module.RealEstate.RealEstateRepository)
    private readonly repository
  ) {}

  @Process()
  async removeEntities(job: Job<IInvalidRealEstate[]>) {
    const { data } = job

    await Promise.all(
      data.map(async ({ source }) => {
        const entity = this.repository.findOne({
          where: [{
            key: 'source',
            value: source,
            operator: '=',
          }]
        })
        await this.repository.delete(entity)
      })
    )
  }
}
```

Listing 5.9: Implementation of the remove estate processor

The *@Processor* decorator is imported from the *@nestjs/bull* package. Using that decorator on top of the class defines that this concrete class can listen to *bull* events and thus process the jobs. The *@Process* decorator function defines that the concrete method is responsible for processing the jobs in the queue, so once the job is inserted into the queue, it will be automatically processed by internally calling this function.

On the other hand, the *NewRealEstateProcessor* is more complex as it must decide whether the received real estate already exists or if it is a duplicate of another real estate.

```
...
@Process()
async processEntities(job: Job<IScraperRealEstate[]>) {
  // The URL of adverts can change in a time
  // this method updates them
  await this.fixUrlChanges(job.data)

  // Convert plain objects to class instances
  const realEstates: Core.Entity.RealEstateEntity[] = await
    this.convertToClassInstances(job.data)

  // Process every single estate
  for (const newEstate of realEstates) {
    // Check if estate exists or not
    const dbEstate = await this.findOriginEstate(newEstate)

    if (!dbEstate) {
      await this.repository.insert(newEstate)
      continue;
    }

    const diff = Core.Util.diffEntities(newEstate, dbEstate)
    if (Object.keys(diff).length > 0) {
      await this.repository.update({
        id: dbEstate.id,
        ...diff,
      })
    }
  }
}
...
```

Listing 5.10: Implementation of the add estate processor

5.5 Web application

The *web* package provides web application that is the only visible part for the end user. The web application is built on *React* UI, the UI components are build on top of *Ant Design*[31] with state managed by *Redux* and side effects (like calling API) handled by *Redux-Toolkit Query*. For SEO markup, the *React-Helmet* library has been used.

The application design is built in a functional matter, rather than object-oriented, as is the case in this project's other back-end application. One of the main reasons for that is that functions can be *tree-shakeable*[32] and thus reduce the bundle size of the application. Besides that, the pure functions are easier to test and easier to use and composite together. From the architectural perspective, the application is built on reusable modules, where each module is oriented on a specific feature and groups the related functionality. Firstly, let us explore the collapsed source directory structure.

```
 index.tsx .................................. application's entry point
 ├─ App.tsx. .................... component for bootstraping core packages
 ├─ utils. ..................................... shared utilities functions
 ├─ config. ........................................... application's config
 ├─ constants. . ................................ application's constants
 ├─ hooks. ................................................ top-level hooks
 ├─ translations. ............................... translations directory
 ├─ pages. ................................... components for page views
 └─ modules
    ├─ adverts. ....................... components for displaying adverts
    ├─ core. ......................... core components for app initialization
    │  ├─ api. ................... services for communication with an API
    │  ├─ store. ........................... application state managment
    │  ├─ i18n. .................... services for translations within the app
    │  ├─ router. ...................... shows a page based on URL path
    │  └─ styles. .......... global CSS resets, external load of CSS modules
    ├─ error. ........................ error components and error handling
    ├─ layout. ......................... layout, header, navigation, footer
    ├─ network. ....................... logic related to network conditions
    ├─ search. .......................... search form, autocomplete fields
    └─ ui. ..................... pure UI components with no-dependencies
```

Figure 5.7: Directory listing of *web* package

[31] https://ant.design/
[32] bundler do not bundle code that you do not use

The most crucial component, which is run first, is *App.tsx*. It simply initializes the external libraries to allow usage of their components inside an application.

```
const App = () => (
  // Initialize application state (store)
  <Redux store={store}>
    {/* Connect app router to the store */}
    <ConnectedRouter history={history}>
      {/* Monitor and notify user about network state changes */}
      <NetworkCheck />
      {/* Initialize styled-components for CSS-in-JS styling */}
      <StyledComponents>
        {/* Initialize translation package */}
        <Lingui>
          {/* Render page layout */}
          <PageLayout>
            {/* Run Router to serve concrete page */}
            <AppRouter />
          </PageLayout>
        </Lingui>
      </StyledComponents>
    </ConnectedRouter>
  </Redux>
)
```

Listing 5.11: *App.tsx* component in web application

Before we further explore the more complex modules, we must stop at the *ui* module. The *ui* module contains only those components which are not related to any concrete feature, like titles, paragraphs, buttons or images. If one wants to change the button's colour, he only wraps the component with customized styles but does not copy the whole component. Those components can be then used as a visual representation of the UI system with libraries like *StoryBook*.

The *AppRouter* component from *"modules/core/router"* listens for URL changes and based on that change renders the appropriate page component from *"pages"* directory. The loading is done using the *React Suspense* feature, which provides a lazy loading mechanism. For instance, when a user visits the detail of a real estate, the browser will not download the components used in a search form and vice versa. Because less data is transferred, less code is executed, leading to an increase in overall application performance. Note that the lazyload procedure comes inside the *AppRouter* component, but the *PageLayout* component is rendered before. In other words, the page layout will be displayed while the appropriate page will be loading.

We place restrictions for page components not to contain any internal state. They should compose other components to render a given page. The reason for the above is that changing the state in a top-level component would cause a re-render to all sub-components.

The following Listing 5.12 shows how the homepage representation looks from the top-level component.

```
const HomePage = () => (
  <>
    {/* SEO related data */}
    <Helmet>
      <title>{t({ id: 'homepage.seo.title' })}</title>
      <meta
        name="description"
        content={t({ id: 'homepage.seo.description' })}
      />
    </Helmet>

    <ImageHeader
      src={'/images/header-hp.jpg'}
      title={<Trans id={'homepage.title'} />}
      description={<Trans id={'homepage.description'} />}
    />

    <StyledContent fullWidth={false}>
      <RecentlyAdded />
      <RecentlyUpdated />
    </StyledContent>
  </>
)
```

Listing 5.12: Preview of *HomePage.tsx* component

One can see that the page component composes the related components together. Using this principle, one can easily locate the needed component. Note, the *Trans* component and *t* macro function, these come from translation library called *LinguiJS*[33]. Based on the given *id* property, the library will find the corresponding translation. Those translations are located in a *"translations"* directory. The current language can be easily changed in the root component and the application will re-render.

The **layout** module includes components related to *Header*, *Footer* and page grid system. Besides that, the module contains functionality related to the side panel.

The **search** module is responsible for all functions related to the search form and searching in general. The main component is *AdvancedSearchForm*, which corresponds to the form in wireframes. Each field shown in the form is a separate component, containing just the necessary actions to display themselves. The search form component composes them together. The field registration to the form state is automatically done via the internal context mechanism provided by *Ant Design*, which we use underlying input components from. Names of fields within the form contain very compact naming. Those namings correspond to *IRealEstateEntity* properties and thus can be easily converted to a requested filter expression depicted in Listing 5.6. For filters with custom comparison ($<, >, <=, >=, IN$) the square brackets are used, for instance, *price*[>=]. Once the filters are validated, they are serialized via *qs*

[33] https://lingui.js.org/

library to URL encoded form and sent to the server and propagated to the browser's navigation bar (as a search parameter). During the client's browsing through the app, thanks to propagating state to the URL, an application can update accordingly even if the user uses the back button in a browser.

The *adverts* module is responsible for everything related to displaying information about adverts, whether a list of them or a single detail. The following Figure shows which components are used when the application displays found real estate.

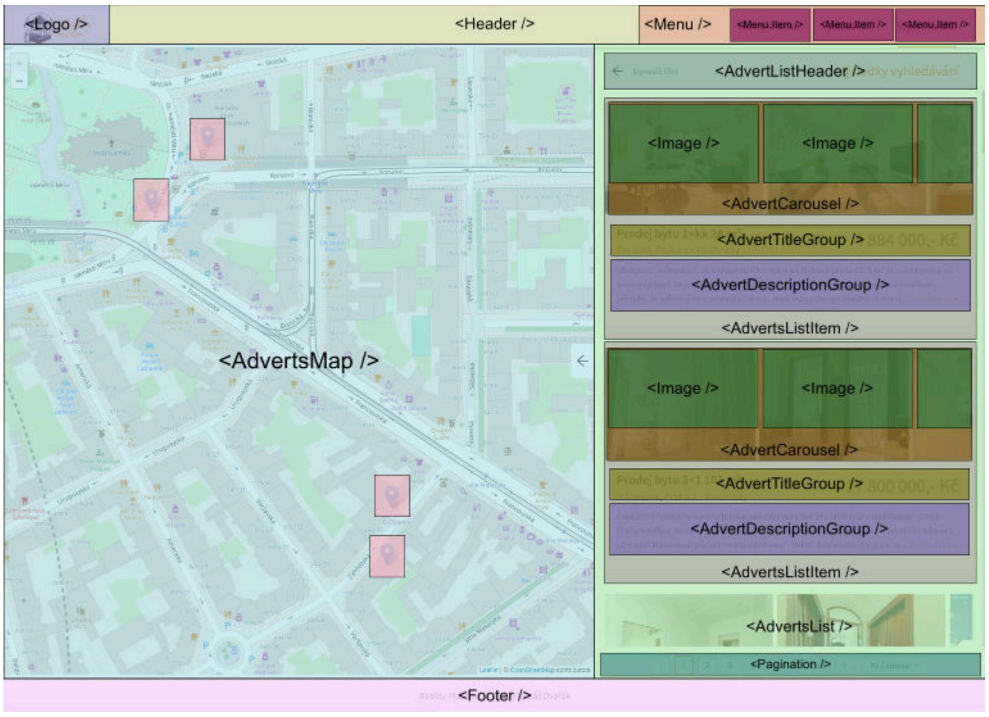


Figure 5.8: Preview of components composition on a listing page

Note that only the most important parts are shown. The internal implementation is even more granular. For map visualization, the *Leaflet* library was used with external plugin *Leaflet.markercluster*. This plugin solves a situation where multiple estates are close to each other. The *AdvertCarousel* handles different sizes of images and adapts accordingly to them. The *Image* component has a fallback mechanism. If the image fetching fails, it displays a placeholder. Based on window size, the page will update accordingly, including the collapse of a map panel or the list panel.

Once the *AdvertsMap* component or *AdvertsList* component mounts, they trigger a request to the API. The underlying mechanism is smart enough to recognize that two components want the same data and sends only one request.

When the same request occurs in a short period, it is retrieved from the local browser cache and no request is sent to the server. Once the advert detail is opened, the following requests are sent:

- get the advert information,
- get 50 closest real estate near me,
- get the information about places nearby (within 500 meters).

Those requests are processed asynchronously and displayed independently. All components which display some data provide a loading placeholder. When such an error occurs, components provide an alert box with the error and *Try again* button. The error message is either a general one or a concrete one received from an API endpoint. Based on the selected language, the API can further translate an error message to a given language.

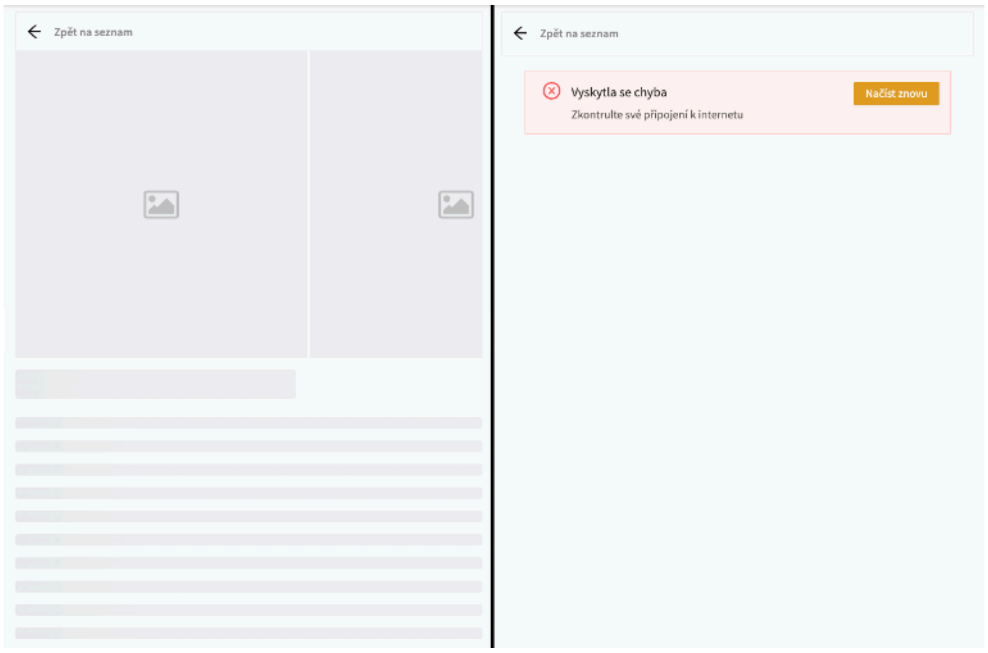
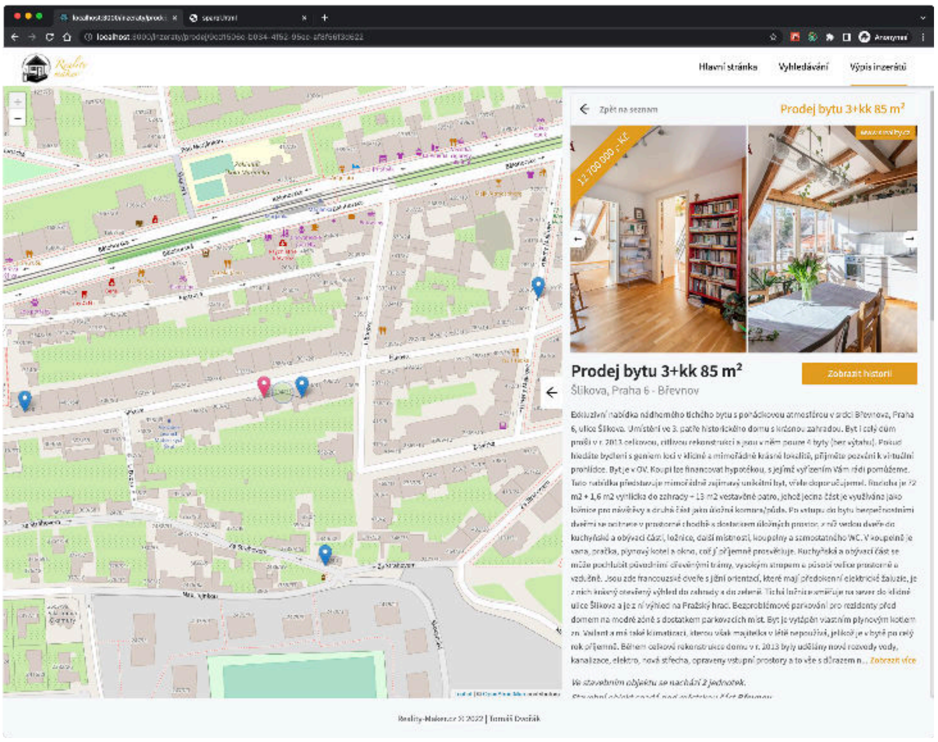


Figure 5.9: Loading and error state preview

The following Figures show an advert detail page, which combines resources from scraped data, RUIÁN and OSM. Place detail combines sources from OSM and Wikidata. Data from OSM serves for displaying names for civic facilities and public transport. RUIÁN data shows additional information about the building that the flat belongs to.



Prodej bytu 3+kk 85 m²

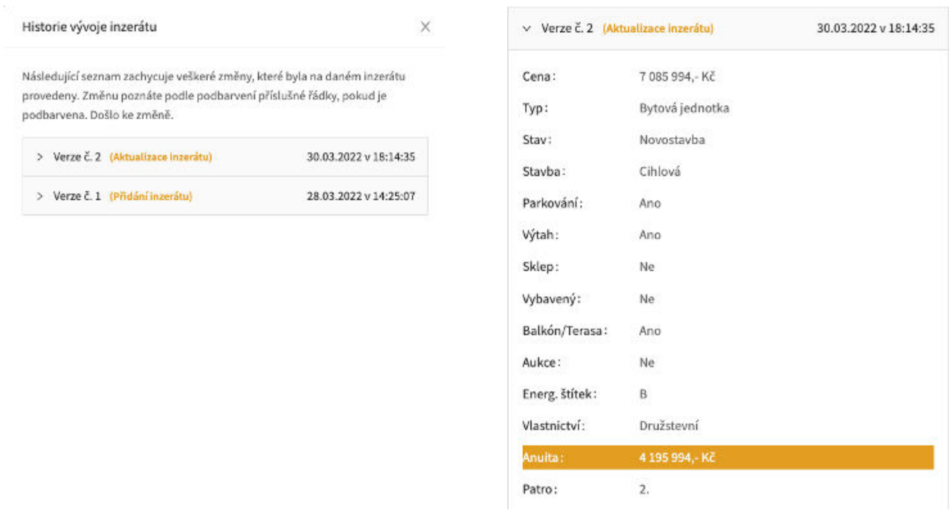
Šlikova, Praha 6 - Břevnov

Exkluzivní nabídka nádherného tichého bytu s pohádkovou atmosférou v srdci Břevnova, Praha 6, ulice Šlikova. Umístění ve 3. patře historického domu s krásnou zahradou. Byt i celý dům prošli v r. 2013 celkovou, citlivou rekonstrukcí a jsou v něm pouze 4 byty (bez výtahu). Pokud hledáte bydlení s geniem loci v klidné a mimořádně krásné lokalitě, přijměte pozvání k virtuální prohlídce. Byt je v OV. Koupi lze financovat hypotékou, s jejímž vyřízením Vám rádi pomůžeme. Tato nabídka představuje mimořádně zajímavý unikátní byt, vřele doporučujeme!. Rozloha je 72 m2 + 1,6 m2 vyhlídka do zahrady + 13 m2 vestavěné patro, jehož jedna část je využívána jako ložnice pro návštěvy a druhá část jako úložná komora/půda. Po vstupu do bytu bezpečnostními dveřmi se ocitnete v prostorné chodbě s dostatkem úložných prostor, z níž vedou dveře do kuchyňské a obývací části, ložnice, další místnosti, koupelny a samostatného WC. V koupelně je vana, pračka, plynový kotel a okno, což ji příjemně prosvětluje. Kuchyňská a obývací část se může pochlubit původními dřevěnými trámy, vysokým stropem a působí velice prostorně a vzdušně. Jsou zde francouzské dveře s jižní orientací, které mají předokenní elektrické žaluzie, je z nich krásný otevřený výhled do zahrady a do zeleně. Tichá ložnice směřuje na sever do klidné ulice Šlikova a je z ní výhled na Pražský hrad. Bezproblémové parkování pro rezidenty před domem na modré zóně s dostatkem parkovacích míst. Byt je vytápěn vlastním plynovým kotlem zn. Vaillant a má také klimatizaci, kterou však majitelka v létě nepoužívá, jelikož je v bytě po celý rok příjemně. Během celkové rekonstrukce domu v r. 2013 byly udělány nové rozvody vody, kanalizace, elektro, nová střecha, opraveny vstupní prostory a to vše s důrazem n... Zobrazit více

Ve stavebním objektu se nachází 2 jednotek.

Reality-Maker.cz © 2022 | Tomáš Dvořák

Figure 5.10: Detail page of real estate (top view)



Historie vývoje inzerátu

Následující seznam zachycuje veškeré změny, které byla na daném inzerátu provedeny. Změnu poznáte podle podbarvení příslušné řádky, pokud je podbarvena. Došlo ke změně.

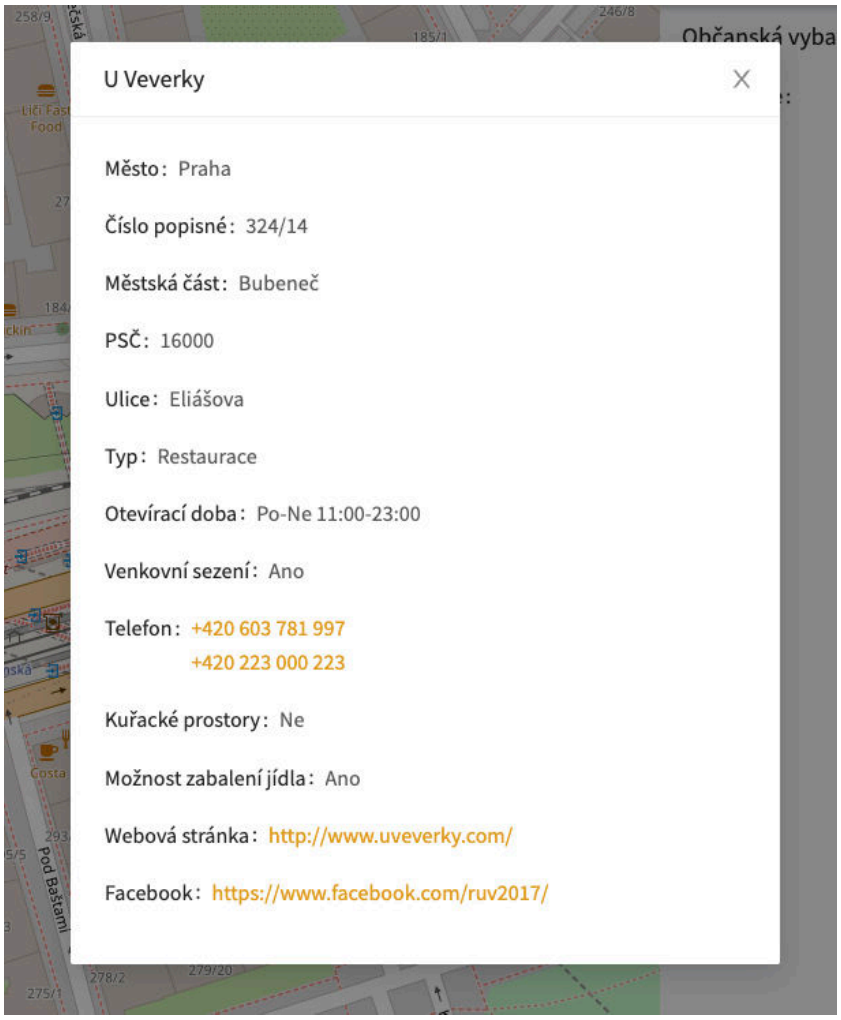
> Verze č. 2	(Aktualizace inzerátu)	30.03.2022 v 18:14:35
> Verze č. 1	(Přidání inzerátu)	28.03.2022 v 14:25:07

Verze č. 2 (Aktualizace inzerátu) 30.03.2022 v 18:14:35

Cena :	7 085 994,- Kč
Typ :	Bytová jednotka
Stav :	Novostavba
Stavba :	Cihlová
Parkování :	Ano
Výtah :	Ano
Sklep :	Ne
Vybavený :	Ne
Balkón/Terasa :	Ano
Aukce :	Ne
Energ. štítek :	B
Vlastnictví :	Družstevní
Anuita :	4 195 994,- Kč
Patro :	2.

Figure 5.11: Preview of history dialog in collapsed and expanded state

Figure (5.12) shows what the modal dialog for a place detail looks like. The received information about a particular place has flexible structure and thus the web application has specific transformers for these fields (phone number, website, e-mail addresses, opening hours and so forth). If there is no existing transformer for a given fact, it will simply render the text information as it was received. Similar process applies to translations. Data about places are retrieved from OSM and *Wikidata*, and they are served from the API.


U Veverky

Město: Praha

Číslo popisné: 324/14

Městská část: Bubeneč

PSČ: 16000

Ulice: Eliášova

Typ: Restaurace

Otevírací doba: Po-Ne 11:00-23:00

Venkovní sezení: Ano

Telefon: +420 603 781 997
+420 223 000 223

Kuřacké prostory: Ne

Možnost zabalení jídla: Ano

Webová stránka: http://www.uveverky.com/

Facebook: https://www.facebook.com/ruv2017/

Figure 5.12: Place detail (OSM & Wikidata)

5.6 API

The *api* package, like other backend services, is based on *NestJS*. Apart from other services, API exposes the interface for communication over HTTP. The API follows *json:api* standard and is built in a MVC style. Computational heavy endpoints use caching mechanism backboned by distributed key-value storage called Redis. The interface provided by an API in the form of HTTP endpoints is fully documented by OAS and accessible through *Swagger UI*.

Before deep-diving into an implementation, let us take a moment on the documentation. The OAS supports two data formats: JSON and YAML [?]. One can first implement an API and then manually create the documentation scheme in one of the mentioned formats. A different approach is the automatic generating, such as in case of the NodeJS project, powered by *TypeScript* decorators applied to entities and DTO classes which are then used to generate the schema. Note that runtime code cannot use interfaces for schema generation, as they do not exist in runtime because JavaScript is a dynamically-typed language. The second approach was chosen and uses helper functions from *@nestjs/swagger* library, which later builds the schema on the fly. Another tooling is located in *"src/swagger"* directory. Once the entity's shape has changed, it will automatically propagate to the schema without additional work. The following Figure depicts the overall directory structure of an *api* package (for the sake of simplicity, some files have been excluded).

```
 main.ts....................................application's entry point
├── bootstrap.ts...........................initializes NestJS application
├── app.module.ts
├── app.controller.ts
├── filter.......................................error exception filters
├── decorator...........................custom application decorators
├── interceptor...........................request/response transformers
├── serializer.....................entity convertor to json:api structure
├── swagger....................utility functions related to documentation
└── modules
    ├── health...................................application's health check
    │   ├── health.module.ts
    │   └── health.controller.ts
    ├── estates
    │   └── ....
    └── places
        └── ....
```

Figure 5.13: Directory listing of *api* package

The *boostrap.ts* besides initializing the application, sets up *CORS* policy, *Swagger UI* documentation, global request/response interceptors, middlewares, pipes and filters.

During the application initialization, the *AppModule* registers database connection (provided from *@reality-maker/core* package). Besides that, it initializes every other module mentioned in the directory listing above.

Once the application is fully initialized, the underlying *Express* server starts listening on a given port (propagated from environment variables).

5.6.1 Estates module

The *estates* module realizes endpoints depicted in diagram 4.13. The module by itself has the following directory structure.

```
estates.moule.ts
├── estates.controller.ts ...................... retrieves HTTP requets
├── estates.service.ts
├── dto. ........................ query parameters representation by classes
│   └── estateQuery.dto.ts
├── transformer. .................... transform query parameters to classes
│   ├── estatesFilter.transformer.ts
│   └── estatesSort.transformer.ts
├── validators. ................ validate received query/search parameters
│   └── estatesQuery.validator.ts
└── serializer. ....................... convert entities to json:api format
    ├── estate.serializer.ts
    └── estateHistory.serializer.ts
```

Figure 5.14: Directory listing of *estates* module inside *api* package

The most complex processing method of the *EstatesController* controller is the method for listing through adverts because it is the endpoint which receives the data from the search form in the web application. Before we look at the given method, we first need to understand how we transform and validate inputs. The core of this information is inside the given DTO, which is later mentioned inside a controller method.

```
class EstatesQueryDto {
  @ApiProperty() // expose to OAS schema
  @ValidateNested() // follow validation rules in sub-class
  page: PaginationDto

  @ApiProperty() // expose to OAS schema
  @IsOptional()
  @EstatesFilterTransformer() // transform before validating
  @EstatesFilterValidator() // custom validation function
  filter: IEstatesQueryFilter = {
    baseFilters: [],
    virtualFilters: {},
  }

  @ApiProperty()
  @IsOptional()
  @EstatesSortTransformer()
  sort: IEstatesQuerySort = []

  @ApiProperty()
  @IsOptional()
  @IsArray()
  @IsIn(AllowedFields, {
    each: true,
  })
  fields: IEstatesFields = []
}
```

Listing 5.13: Preview of real estate DTOs filter.

```
  ...
  //  handles root path of "/estates" resource
  @Get()
  // hint OAS generator
  @ApiProducesJsonApi()
  @ApiNestedQuery(EstatesQueryDto)
  public async findAll(@Query() filters: EstatesQueryDto) {
    // delegate request processing to the service
    const { response, count } = await this.service.findAll(filters)

    // transform to json:api format
    return this.serializer.serializeAsync(response, {
      count,
      search: filters,
    })
  }
  ...
```

Listing 5.14: Controller method for real estate listing

As one can see, using TypeScript decorators leads to an elegant and declarative way to extend functionality. Without them, the controller method or service must handle the validation process. In our case, we tell the framework that we expect the query filter as an instance of *EstatesQueryDto* class, and if that conversion is not possible, throw an error[34]. The internal logic of how the validation and transformation work is not described because they it is not crucial for understanding the overall data flow. The only important thing about them is that they convert the inputs to the object shape, which can be directly passed to the entity repository. They also split basic filters related to entity and filters related to social amenities and public transport. The next part describes how the filters are evaluated and processed in the *EstatesCacheService*.

[34]framework is smart enough to throw *BadRequestExpection* with the correct status code (400) rather than *InternalServerError* with status code 500.

The *EstatesCacheService* is a wrapper around *EstateService*, where *EstatesCacheService* overrides particular methods, where if the cache entry exists, it returns it. Otherwise, it fallbacks to the base service and then saves the result to the cache. The dependency injection mechanism resolves to the concrete one based on the environmental variable. If the caching is enabled, the resolution token for *EstateService* will resolve to *EstateCacheService*. With this solution, no “if“ statements are needed, and the code is thus cleaner.

Based on the received filters, the service prepares the filters and sends two asynchronous requests to the database, one for the data and one for the count of total results to provide data for pagination on the web. Some filters take precedence to optimize the query performance. For instance, comparing the distance between two points or checking if the given point lies in a given polygon or even multi polygon is much more expensive than comparing if the flat's floor area is over $50m^2$. The subsequent optimization is to split the query into two parts:

1. retrieve estates matching the filters from graph A,
2. retrieve social facilities and public transport from graph B.

To imagine how such a SPARQL query looks, let us show one. The following list will define our requirements:

- price must be less or equal to 7 500 000, Kč,
- usable area at least $50m^2$,
- building should be made of brick or stone,
- should lie in a ”Dejvice” district,
- the layout of the flat should be one of the following 2+1 or 2+kk,
- at least one subway stop must be within 500 metres,
- at least one restaurant must be within 250 metres,

Those requirements can be quickly filled inside a search form, sent to the API, transformed to a given shape, validated, retrieved in a controller, passed to service and finally composed together in a repository. The following SPARQL query represents the retrieved filters.

```
SELECT DISTINCT ?id ?usableArea ...
WHERE {
  GRAPH <http://reality-maker.cz/graph/sell> {
    ?entity a realm:RealEstate ;
      dcterms:identifier ?id ;
        realm:usableArea ?usableArea ;
        realm:layout ?layout ;
        wgs:lat ?mapLat ;
        wgs:long ?mapLon ;
        realm:price ?price ;
        realm:structure ?structure .

    FILTER (?price <= "7500000"^^xsd:nonNegativeInteger)
    FILTER (?layout IN ("2+1", "2+kk"))
    FILTER (?structure IN ("Cihlová"@cs, "Kamenná"@cs))
    FILTER (?usableArea >= "50"^^xsd:positiveInteger)

    # The "Dejvice" district
    BIND("MULTIPOLYGON(((14.4075807 50.097657...)))"
         ^^virtrdf:Geometry as ?districtBoundary)
    BIND(bif:st_point(?mapLon, ?mapLat) as ?mapPoint)
    FILTER(bif:st_intersects(?districtBoundary, ?mapPoint))
  }

  GRAPH <http://reality-maker.cz/osm/cze/prague> {
    # Has restaurant within 250 meters
    FILTER EXISTS {
      ?s rdf:type osm:node ;
         geo:hasGeometry ?geo ;
         osmt:amenity "restaurant" .
      MINUS {
        ?s rdf:type osm:node ;
           geo:hasGeometry ?geo ;
           osmt:amenity "restaurant" .

        FILTER(geof:distance(?mapPoint, ?geo, uom:metre) > 250)
      }
    }

    # Has subway within 500 meters
    FILTER EXISTS {
      ?s rdf:type osm:node ;
         geo:hasGeometry ?geo ;
         osmt:public_transport "station" ;
         osmt:railway "station" ;
         osmt:subway "yes" .

      MINUS {
        ?s rdf:type osm:node ;
           geo:hasGeometry ?geo ;
           osmt:public_transport "station" ;
           osmt:railway "station" ;
           osmt:subway "yes" .

        FILTER(geof:distance(?mapPoint, ?geo, uom:metre) > 500)
      }
    }
  }
}
LIMIT 10
```

Listing 5.15: Representation of complex filter in a SPARQL query

Note that the filters used represents only a tiny subset of all possible combinations. If one wants to use more filters for public transport or amenities, another block of expression must be used. Otherwise, they will be executed as logical *OR* rather than logical *AND*. One can note that those *MINUS* blocks can be omitted entirely and replaced by a single distance filter; the reason why they are not is based on the Virtuoso evaluation. Without the *MINUS* block, the evaluation time grows exponentially, whereas, with the selected approach, the time grows linearly. The reason for the slow evaluation is probably because the variable will be bounded, and thus the *FILTER* operation will cause a range scan over the whole graph (internally *table*).

To support the filtering by the UTF8 characters, one must compile the latest Virtuoso from the develop branch due to issues with UTF8 encoding in the stable version. After that, one must set *XAnyNormalization = 3* inside the Virtuoso configuration file and enable *geos* plugin for supporting geometry functions. The final configuration file used is located within the project.

Once the query execution is complete, the cache service will create a hash from the filter object and store it to Redis. The cache entry expiration is set to one hour and can be easily changed.

Other endpoints are based on similar logic, except they do not receive filters from the user, apart from query parameters used for selecting a single entity.

5.6.2 Places module

The *places* module is structured in a similar fashion as the *estates* module. The main responsibility of the *places* module is to provide information about given places identified by an OSM identifier. Besides that, the module provides an endpoint for search by providing just a part of the district's name.

```
│ places.module.ts
├─places.controller.ts
├─places.service.ts
├─places.serializer.ts
└─utils
   └─amenityTypeConvertor.ts
```

Figure 5.15: Directory listing of *places* module inside *api* package

As mentioned in the Chapter 4 (Design), the *places* controller exposes two following endpoints. One for receiving data and the second for retrieving information about a given place. Once the request is retrieved inside a controller, it is then delegated to the *PlaceService*, the results are then passed to the *PlaceSerializer* which converts the output to the *json:api* data shape.

Following SPARQL query shows how the retrieval of district names is done, not that the graph name differs. This is due to improving query execution.

```
SELECT DISTINCT ?name
FROM <http://reality-maker.cz/osm/cze/prague/districts>
WHERE {
    ?iri a osm:relation ;
        osmt:type "boundary" ;
        geo:hasGeometry ?placeGeo ;
        osmt:name ?name .

  FILTER(bif:contains(?name, '"Dejv*"'))
}
ORDER BY ?name
```

Listing 5.16: SPARQL query for searching city districts

The following SPARQL query shows how to retrieve the nearest tram stop for a given point in a radius of 500 meters. For trams and bus stops, one can receive duplicated names. That is because there are more platforms with the same names. This issue is solved by picking the one closer to the given point.

```
SELECT DISTINCT ?iri ?name ?distance ?color
FROM <http://reality-maker.cz/osm/cze/prague>
WHERE {
    ?iri a osm:node ;
        osmt:name ?name ;
        osmt:tram ?type ;
        geo:hasGeometry ?geo .

    FILTER(REGEX(?geo, "^POINT" ))
    BIND(bif:st_distance(?geo, bif:st_point(...)) as ?distance)
    FILTER(?distance < "5.0E-1"^^xsd:double)

  FILTER NOT EXISTS {
    ?iri2 a osm:node ;
      osmt:name ?name ;
      osmt:tram ?type ;
      geo:hasGeometry ?geo2 .

    FILTER(REGEX(?geo2, "^POINT" ))
    BIND(bif:st_distance(?geo2, bif:st_point(...)) as ?distance2)
    FILTER(?distance2 < ?distance)
  }
}
ORDER BY ?distance
```

Listing 5.17: SPARQL query for finding the closest tram stations

To retrieve data from RUIÁN, we first need to find the related reference. The input for this query are coordinates of the real estate.

```
PREFIX ruian: <https://ruian.linked.opendata.cz/slovník/>
PREFIX schema: <http://schema.org/>
PREFIX osmt:  <https://www.openstreetmap.org/wiki/Key>

SELECT DISTINCT ?flatCount ?floorCount ?district
WHERE {
    ?s rdf:type osm:way ;
      osmt:ref:ruian:building ?ruian ;
      geo:hasGeometry ?geoBuilding .

    FILTER(REGEX(?geoBuilding, "^MULTIPOLYGON"))
    FILTER(bif:st_intersects(?geoBuilding, bif:st_point(...)))

    BIND(IRI(CONCAT(
      'https://ruian.linked.opendata.cz/zdroj/stavební-objekty/',
      ?ruian)) as ?ruianIRI)

    SERVICE <https://ruian.linked.opendata.cz/sparql> {
      OPTIONAL { ?ruianIRI ruian:početBytů ?flatCount . }
      OPTIONAL { ?ruianIRI ruian:početPodlaží ?floorCount . }
      OPTIONAL { ?ruianIRI ruian:částObce/schema:name ?district . }
    }
}
```

Listing 5.18: SPARQL Federated query for retrieving RUIÁN data based on the OSM reference

The reason for queries to external services being split is due to performance. Querying public SPARQL endpoints are not processed within milliseconds but rather in seconds.

Chapter 6

Testing

The following Chapter describes the overall testing of the client-side and server-side of the application. The testing process starts with **Automated testing**, where each application or even shared package contains test cases, which ensures that the given functionality behaves as expected. On the other hand, the second part of the testing process is based on **User testing**, which rather reveals the usability problems. For unit and integration testing, the *Jest* [48] library was selected. E2E tests were done by *Cypress* [49].

6.1 Unit Testing

The idea behind unit testing is to test a small isolated unit, ideally a single function[35]. Those units are isolated from the surrounding code by *mocking* their external dependencies. [50]

The unit tests were created for all packages except for *shared* package (no functionality, just types and enums) and *web* application (will be tested later by E2E testing). In every package, unit tests are located within the given module and their filename has suffix *"*.spec.ts"*. The following listings showcase some of the unit tests used in this project.

```
it('On insert only beforeInsert is called', async () => {
  subscriber.startListening()

  const spy = jest.spyOn(subscriber, 'beforeInsert').mockImplementation()

  await repository.insert(validRealEstate)
  expect(spy).toBeCalledTimes(1)
})
```

Listing 6.1: Verify that the repository will call *beforeInsert* method on the subscriber (*core* package)

[35]In OOP world, just a single method

The Listing 6.2 verifies that the valid real estate object can be transformed to the corresponding RDF and vice versa.

```
it('Construct Class from RDF and vice versa', async () => {
  const target: RealEstateEntity = fromPlainToEntity(
    RealEstateEntity,
    createValidRealEstate()
  )
  const { id } = target

  const targetRDF = EntityTransformer
    .toRDF(id, target, RealEstateEntity)

  const targetB = EntityTransformer.fromRDF(RealEstateEntity, targetRDF)
  const targetBRDF = EntityTransformer.toRDF(id, target, RealEstateEntity)

  expect(targetB).toStrictEqual(target)
  expect(targetRDF).toStrictEqual(targetBRDF)
})
```

Listing 6.2: Transform RDF to Class and vice versa (*core* package)

To showcase a test case from a different package, let us show a test case responsible for parsing the advert from *Bezrealitky* portal (*scraper* module). Because of making unit testing and not E2E test, we mock the server response and thus verify only the parsing process.

```
it.each(['715060', '694271', '678645', '709843'])('Parsing %s',
  async (advertId: string) => {
    // Because of unit testing, mock server response
    const html = await fs.promises.readFile(
      path.join(__dirname, `__fixtures__/${ID}.fixture.html`)
    )
    // Mock response to HTTP call that the parser will trigger
    fetchMock.mockOnce(html.toString())

    const estate = await service.parse(advertId)

    // Verify it matches the expected snapshot
    expect(estate).toBeDefined()
    expect(estate).toMatchSnapshot({
      sourceUpdatedAt: expect.any(Date),
    })
  }
)
```

Listing 6.3: Parsing the advert from *BezRealitky* portal (*scraper* package)

In summary, the whole system contains over **60** unit test cases, where most of them belongs to *core* and *scraper* package. The reason why the *core* module is highly tested is that other modules directly depend on it as it contains repositories and database connections.

6.2 Integration testing

In unit tests, we were trying to verify that a small unit of codes behaves as expected [50]. However, widespread practice is that multiple distinct parts of the application communicate between themselves, and we want to verify that the cooperation between those parts is working. Having just unit tests, we would not know until we go to production that the system as a whole is not working. On the other hand, having just integration tests, we would know that the system is broken, but we would not know where. That is why it is beneficial to have both of them.

In integration tests, we test larger parts of the application that somehow communicate together [50]. The next crucial part is trying to minimize the need for mocking dependencies. It can be, for instance, executing SPARQL queries against the database rather than mocking the responses.

Because of testing bigger parts of functionality which is not always within a single module, the test cases are located within *src/test* directory in each tested package. The filename of those tests has suffix *".int.spec.ts"*.

Listing 6.4 depicts how the single entity is scraped and passed to the queue for the next processing. At this point, the responsibility of the *scraper* package ends. Following processing will be done in *analyzer* package. Note that we have to mock the server response for the advert because the advert can be no longer available - it is either deleted or obsolete (as the property is already sold). Listing 6.5 depicts the test case of how the *analyser* package consumes the jobs from the queue.

```
it('Scrapes single entity and add it to the queue', async () => {
  const data = await fs.promises.readFile(path.join(
    __dirname, `../__fixtures__/715060.fixture.html`
  ))
  fetchMock.mockOnce(data.toString())

  // Retrieve queue job created by the CoreService
  const { added, removed } = await service.revalidateOne(
    'https://www.bezrealitky.cz/nemovitosti-byty-domy/715060...'
  )

  // Nothing was inserted into the delete queue
  expect(removed).toBeNull()
  await expect(removeQueue.count()).resolves.toBe(0)

  // Only one job with one advert has been added to the new queue
  expect(added).toBeDefined()
  expect(added.data).toHaveLength(1)
  await expect(addQueue.count()).resolves.toBe(1)

  await added.remove()
})
```

Listing 6.4: Verify parsing and ability to add advert to the queue for next processing (*scraper* package)

```
it('Receive and insert a real estate to the database', async () => {
  // Verify that the database is empty
  await expect(repository.count()).resolves.toBe(0)

  // Simulate the situation, that somebody adds a job to the queue
  const job = await addQueue.add([scraperResult], {
    removeOnComplete: true,
    removeOnFail: true
  })

  // Wait until the job is fully consumed by the producer
  await job.finished()

  // Verify that the real estate has been saved to the database
  await expect(repository.count()).resolves.toBe(1)
})
```

Listing 6.5: Verify that the processor consumes the job and saves the real estate to the database (*analyser* package)

Besides that, the *analyzer* package contains additional test cases, including insertion of invalid real estate, handling URL changes, processing jobs from the delete queue and so forth. As previously mentioned, most of the test cases are contained within *core* package, which contains crucial services for working with the database.

6.3 E2E Testing

End-to-end (E2E) tests the functionality of the whole system and can be done in several ways. Those tests reveal if the system is working as expected from the client way, where the client can be a web application or application calling the exposed API. [50]

Tests were made for the web application (*web* package) because it tests the system from various perspectives. First, it tests that the UI of the web application is working, and secondly, it verifies that the API is working and provides data in the required shape.

Because the web application is implemented as SPA, we need to run a headless browser which we can control via *JavaScript*. This functionality is provided by *Cypress* library [49]. *Cypress* testing syntax is similar to syntax that *Jest* uses. Test cases are located in *Cypress* directory inside the *web* package. Besides that the directory contains configuration files and plugins related to the *Cypress*.

Test cases cover all pages that the web application provides. Testing aims to verify that the pages retrieve data from API, react to user inputs and render all the expected components. For this type of testing, no mocks are present. Thus the API and web application must be running. The following figures depict some of the test cases that were used in part of the E2E testing.

```
describe('Adverts page', () => {
  beforeEach(() => {
    // Catch all request to the API for later usage
    cy.intercept('GET', '**/estates?*').as('getEstates')

    // Visit the listing page
    cy.visit('/inzeraty/prodej')
  })

  it('Contains map with markers, list and pagination', () => {
    // Wait and verify server response
    cy.wait('@getEstates')
      .its('response.statusCode')
      .should('be.oneOf', [200, 304])

    // Verify that application renders adverts to the list
    cy.get('[data-cy="advert-list-item-card"]')
      .should('have.length.at.least', 10)
      .as('cards')

    // Verify that every item has a galery
    cy.get('@cards')
      .find('.slick-list')
      .children()
      .should('have.length.at.least', 1)

    // Verify that map is present
    cy.get('[data-cy="map-container"]')
      .should('have.length', 1)
      .should('be.visible')

    // Verify that adverts are also displayed on a map
    cy.get('[data-cy="map-container"]')
      .should('have.length', 1)
      .get('.leaflet-marker-icon')
      .should('have.length.at.least', 10)
      .should('be.visible')

  // Check that pagination is present
    cy.get('.ant-list-pagination')
      .should('have.length', 1)
      .should('be.visible')
  })
})
```

Listing 6.6: E2E test case to verify functionality of real estate listing page (*web* package)

Besides verifying that specific components are present and rendered without an error, other test cases in a given test file focus on a more granular functionality. The example can be verifying that changing the URL parameter for page size propagates to the number of results in a list. Next, it verifies that clicking on the *Next page* button reloads the page content and modifies the URL.

```
describe('Search page', () => {
  beforeEach(() => {
    cy.visit('/vyhledat')
  })

  it('Search for a given district', () => {
    cy.get('[data-cy="search-form"]')
      .should('have.length', 1)

    cy.get('[data-cy="district-autocomplete"]')
      .should('have.length', 1)

    cy.intercept('GET', '**/district?*')
      .as('getDistricts')

    cy.get('[data-cy="district-autocomplete"]')
      .scrollIntoView()
      .type('Dejvice', {
        // simulate slow typing
          delay: 75,
      })

    cy.wait('@getDistricts')
      .its('response.statusCode')
      .should('be.oneOf', [200, 304])

    cy.get('.autocomplete-dropdown')
      .should('be.visible')
      .find('.ant-select-item')
      .should('have.length', 1)
      .eq(0)
      .should('have.text', 'Dejvice')
  })
})
```

Listing 6.7: E2E test case to verify functionality of district autocomplete input (*web* package)

The test cases related to a *Search page* besides the autocomplete feature verify that one can submit a form without explicitly filling the fields, that specific fields cannot contain negative values or that a reset button resets the form to the initial state.

Other pages are tested in the same fashion, starting with checking for all expected components and then testing the individual functionality. Finally, the automated tests cover the functionality requirements of the system defined in Chapter 2 (Requirements).

6.4 User Testing

User testing is a process of testing the user interface and functionality of a given application by allowing real users to perform specific tasks under realistic conditions [51]. In contrast to automated testing, real users can reveal the usability issues rather than the technical issues. The testing process of the web application consisted of three parts:

- **Pre-testing questions** aim to reveal the background of the tester, starting with a set of questions about their knowledge, about the problem that the application solves and also their experience with related applications.
- **User testing** is the part where the tester works on given tasks, and the supervisor takes notes about the user's progress.
- **Post-testing questions** capture the testers' feedback on the system as a whole. They identify why the tester engaged or disengaged with the application and tester's problems with the application.

To determine the ideal number of testers which balances the percentage of founded usability problems and the number of testers needed, the *Nielsen's usability curve* has been used. Based on the curve depicted in Figure 6.1 we can observe that for discovering 85% of the usability problems, we only need 5 testers, whereas, for 100%, that number of testers must be 15 [52]. Based on that claim, **five testers** were invited. The test group was selected to cover both young and middle-aged individuals living in *Prague*.

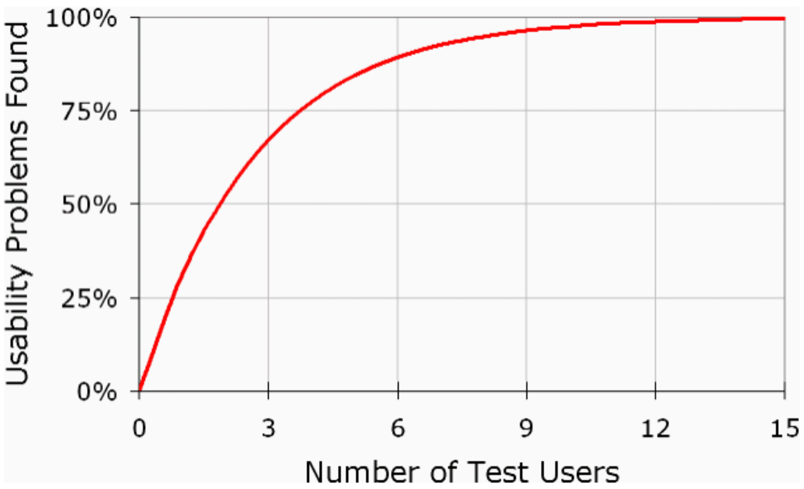


Figure 6.1: Nielsen's usability curve depicting the relation between number of testers and problems found [52]

6.4.1 Pre-test

Before the application testing, every tester was briefly introduced to the application, and then the tester was asked to answer a set of questions describing his background. Answers are depicted in Table 6.1.

Table 6.1: Background information of the tester group

Tester	A (student)	B (student)	C	D	E
Age	24	22	48	39	31
Gender	Male	Female	Male	Female	Female
Field	Economy	BME[36]	IT	IT	Architecture
Education	Bachelor	Bachelor	A level	HND[37]	Master's

After the introduction part, the testers were asked the following set of questions related to real estate:

- Are you interested in the real estate market?
- Have you ever bought or sold real estate?
- Which real estate portals do you know?
- Do you miss any feature on those portals?
- What are the deciding factors when buying real estate?
- If you ever consider buying real estate, would you rather use a real estate portal or engage a real estate agency?

All of the testers share a passive interest in the real estate market. Two of them have sold, bought or rented at least one flat, and they occasionally visit real estate portals to monitor price trends.

One of the testers confirmed missing a filter for adverts without exactly specified price[38] while another tester reported missing more filters about social facilities. The rest of the testers found the set-up satisfactory. All respondents stated that they know *SReality* and *BezRealitky*, two of them also named *M&M reality*[39] and *Reality IDNES* and one named *České Reality* and *RealityMIX*[40].

Concerning the deciding factors, everyone agreed that the most important part is the location with wide availability of civic amenities, followed by public transport. One also mentioned the importance of a number of flats in the

[38]some real estate agencies provide the price only to those who call
[39]https://www.mmreality.cz/
[40]https://realitymix.cz/

building, which is supposedly not provided in any well known real estate portal. Those who never bought a real estate would rather use real estate agency services, whereas those who already bought one would rather do it themselves.

6.4.2 Testing scenarios

The actual testing is done in such a way that the task given to the testers was able to be completed without any additional help or hint. During the execution of the given scenario, notes were taken. Three testers used their own computers while two used the provided laptop. For this user testing session, the web application has been deployed to the private server.

Table 6.2: **Scenario A - Find specific real estate**

Description	Verify the usability of the search form and listing page
Assignment	Find and select one of real estate in “Vinohrady“ district with subway within $350m^2$, price bellow 10 000 000 CZK, “Private“ ownership and is situated on the second floor.
Expected behaviour	1. use the Action button on the homepage or link in the menu, 2. fill a search form, 3. click on one of the adverts in the list.

Observation: Testers successfully completed the task without any problems. Two of them found it challenging to fill the *floor* field to match only those on a second floor.

Performed changes: Based on the issue identified above, the placeholders were added to the search’s form fields.

Table 6.3: **Scenario B - Find specific real estate**

Description	Verify the visibility of clickable place details' links inside the advert detail
Assignment	Find real estate with a restaurant within 500 meters in the “Dejvice“ district and show the detail of such a restaurant.
Expected behaviour	1. fill a search form, 2. click on one of the adverts in the list, 3. scroll down to the “Restaurants“ subsection and click on one of the provided restaurants' names.

Observation: The testers successfully completed the task without any problems, except for one tester, who tried to search the restaurant name on the map without going into detail. Eventually, the tester was asked to find a different way and after a while, managed to locate it in the advert's detail page.

Performed changes: Based on one tester's different approach, the maximal possible zoom settings of the map have been changed.

Table 6.4: **Scenario C - Filter editing**

Description	Verify that tester navigates through the application via provided UI elements.
Assignment	Find real estate with a floor area over 75m^2 in the “Žižkov“ district. Afterwards, change filters to select only those that have a lift.
Expected behaviour	1. click on a search button in the navigation or on the homepage's header, 2. fill the form fields, 3. submit the form, 4. click “Edit filters“ button in the list's header, 5. update the form fields and submit the form again.

Observation: Three out of five testers clicked on the expected “Edit filters“ button, whereas two of them used the web browser’s back button. Besides that, one tester complained about not being able to click on the text but had to click on the back arrow instead.

Performed changes: Because the web application is aware of the web browser’s back button, only the text field has been changed to be also clickable.

Table 6.5: **Scenario D - Pagination and browsing**

Description	Verify that the pagination and its settings are visible.
Assignment	Show list of real estate and go to the next page, then change the pagination size to 20.
Expected behaviour	1. click on “Listing of adverts“ item in the main menu, 2. click on the next page button in the pagination section, 3. wait for the page to load, 4. click to pagination’s size dropdown, 5. select a given page size.

Observation: Testers successfully completed the task in an expected way, except for a little deflection, where one tester went to the listing page through the search form instead of from the main menu, but the result was the same. After that, one of the testers tried to edit filters and noted that his selected page size was not carried through across the filter’s changes.

Performed changes: The state mechanism has been updated to preserve the user’s selected page size.

Table 6.6: **Scenario E - History of a real estate**

Description	Verify that the “Show history“ button on the adverts’ detail is visible and highlighted changes are easy to understand.
Assignment	Select one of the recently updated adverts and investigate the changes made to the advert.
Expected behaviour	1. scroll the page to the section “Recently updated“, 2. select one of the listed adverts, 3. click on the “Show history“ button, 4. collapse the latest version’s changes, 5. note the changed properties.

Observation: Testers completed the task without any problems. However, one tester compared the changed fields with those on the advert’s page because he opened the first revision with the initial version instead of the last revision. Thanks to that, the tester revealed that the price information is missing from the list of advert’s properties. A very surprising finding was that nobody read the short explanation about how the history dialogue works.

Performed changes: The advert’s price has also been added to the advert’s properties table.

6.4.3 Post-test

After the completion of the mentioned scenarios, testers were given the following questions:

- Rate the functionality aspect on a scale of 1 (worst) to 10 (best).
- Rate the visual aspect on a scale of 1 (worst) to 10 (best).
- Rate the usability aspect on a scale of 1 (worst) to 10 (best).
- How would you describe your overall experience with the application?
- What did you like the most?
- What did you like at least?

Table 6.7 depicts responses to first three questions. Testers were overly satisfied with the providing functionality, design and also with the usability.

Table 6.7: Usability post-test rating evaluation

Tester	A	B	C	D	E	Average
Functionality	10	9	9	10	10	9.6
Design	10	10	10	10	9	9.8
Usability	9	9	10	10	9	9.4
Average	9.7	9.3	9.7	10.0	9.3	

They stated that the application is fast and reliable, and the design is pleasing. One tester especially highlighted that his most liked functionality was the auto-scrolling[41]. Others mentioned the possibility of filtering by many types of civic amenities and the ability to display their detail with additional information. Besides the civic amenities, testers appreciated the advert's history feature, which they stated is especially useful for monitoring price changes. The following issues were reported besides the positive feedback:

- cannot zoom the map to the level where one can read the names of the map's objects,
- pagination setting is not preserved after filters change,
- missing placeholders in a search form,
- missing advert's price in a property description,
- text overlaps on a smaller screen.

Those issues were, after the testing session, fixed and deployed. Overall, user testing has been beneficial as it leads to application improvements.

[41]Whenever the user hovers the mouse over the marker on the map, the application will scroll down to the advert detail in a list

Chapter 7

Conclusion

The goal of this thesis was to create a **real estate portal capable of providing advanced search abilities with the use of semantic web technologies and open linked datasets**. The theme was driven by the increase in demand for real estate as result of the COVID-19 pandemic and rising inflation. As the existing real estate portals provide no or limited information about the property's environment, people are forced to browse through various adverts' listings without the ability of more focused personalized searching options (such as civic amenities, public transport etc); for those they have to search separately.

The thesis started with the analysis of a selection of the most popular real estate portals in the Czech Republic and relevant datasets. That was followed by the design and implementation of the web application, which retrieves data from REST API that combines various data sources, starting with extracted data from real estate portals followed by data from OpenStreetMap, Wikidata and RUIÁN. Besides the data extraction, the system periodically monitors new adverts and tracks the changes made to existing extracted details. Those changes are then visible in the web application, where the user can browse through the history. Beyond the scope of the assignment, the application was deployed to the server and made available to the testers.

At the moment, the application provides more search possibilities in terms of comparison with mentioned real estate portals, especially in terms of civic amenities. In terms of the potential production use, the triple-store database would need much more system resources[42], because all of the expensive geometric operations are done on the fly or by additional SPARQL query optimization.

The project architecture is designed as a group of independent and high-scalable applications, ready for running in the Cloud environment. The codebase is purely written purely in TypeScript and uses the Virtuoso triple-store

[42]especially more than the average laptop offers

for data storage. The project code is managed as a monorepo, where each application uses the functionality from the shared package. The system and its sub-packages are tested via automated tests - unit, integration and end-to-end tests followed by user testing. Last but not least, all system parts are fully Dockerized. Besides Docker, the application can be run with the preconfigured PM2 process manager.

During the time spent on this thesis, eight bugs[43] were found in the Virtuoso database. Some of them influence the querying style while others impact the performance or even the matched results. Besides that, other minor issues were reported to the other related libraries and projects.

7.1 Future work

Based on storing all data in *RDF* there are numerous ways how to query them. One can use their datasets with links to *Wikidata*, *DBpedia* or *OpenStreetMap* and thus retrieve data about real estate. For instance, one can ask questions like: *"Give me all real estate within one kilometre of the football stadium, which has more than 500 seats and where at least one team plays with a goalkeeper not born in the Czech Republic, but in a country with the population of over 10 million"*. The query will be complex, but one of the future features can be providing the possibility to enter or even generate SPARQL queries while preserving the provided user interface. Another feature could be scraping data from more real estate portals and removing the restriction on scraping only flats in Prague, which was done due to performance issues.

In addition to further expanding datasets and performance, the web application could offer the following features:

- user registration with the ability to save favourite adverts,
- notify users about changes to their selected real estate,
- provide subscription for newly listed adverts,
- calculate travel distance from given real estate to the user-defined place,
- include other forms of criteria such as rent or roommate.

[43] `https://github.com/openlink/virtuoso-opensource/issues?q=is:issue+author:Tomas2D+`

Bibliography

[1] Výroční zpráva ČNB za rok 2020. [online], Apr 2021, [visited on 2022-04-02]. Available from: https://www.cnb.cz/export/sites/cnb/cs/o_cnb/.galleries/hospodareni/vyrocni_zpravy/download/vyrocni_zprava_2020.pdf

[2] Výroční zpráva ČNB za rok 2021. [online], Apr 2022, [visited on 2022-04-02]. Available from: https://www.cnb.cz/export/sites/cnb/cs/o_cnb/.galleries/hospodareni/vyrocni_zpravy/download/vyrocni_zprava_2021.pdf

[3] Frequently asked question. [online], 2021, [visited on 2022-02-28]. Available from: http://www.cz.gemius.com/caste-dotazy.html

[4] Czech - Gemius Rating. 2021, [visited on 2022-02-28]. Available from: https://rating.gemius.com/cz/tree/2

[5] O službě Sreality.cz. [online], [visited on 2022-03-02]. Available from: https://napoveda.seznam.cz/cz/sreality/o-sluzbe-sreality.cz/

[6] MAFRA, a. [online], [visited on 2022-03-02]. Available from: https://reality.idnes.cz/

[7] INTERNET, [online], [visited on 2022-03-03]. Available from: https://www.ceskereality.cz/

[8] Bezrealitky. O portálu Bezrealitky. [online], [visited on 2022-03-04]. Available from: https://www.bezrealitky.cz/informace/o-nas

[9] W3C. Semantic Web. [online], [visited on 2022-03-10]. Available from: https://www.w3.org/standards/semanticweb/

[10] W3C. RDF 1.1 Concepts and Abstract Syntax. [online], [visited on 2022-03-12]. Available from: https://www.w3.org/TR/rdf11-concepts/

[11] W3C. RDF 1.1 Primer. [online], Jun 2014, [visited on 2022-03-15]. Available from: https://www.w3.org/TR/rdf11-primer/#section-triple

[12] W3C. RDF Schema 1.1. [online], Feb 2014, [visited on 2022-03-16]. Available from: https://www.w3.org/TR/rdf-schema/

[13] W3C. OWL 2 Web Ontology Language. [online], Dec 2012, [visited on 2022-03-20]. Available from: https://www.w3.org/TR/owl-overview/

[14] Berners-Lee, T. Linked data. [online], Jul 2006, [visited on 2022-03-24]. Available from: https://www.w3.org/DesignIssues/LinkedData.html

[15] W3C. SPARQL 1.1 Overview. [online], Mar 2006, [visited on 2022-03-26]. Available from: https://www.w3.org/TR/sparql11-overview/

[16] W3C. SPARQL 1.1 Query Language. Mar 2013, [visited on 2022-03-27]. Available from: https://www.w3.org/TR/sparql11-query

[17] W3C. SPARQL 1.1 Federated Query. [online], Mar 2013, [visited on 2022-03-27]. Available from: https://www.w3.org/TR/sparql11-federated-query/

[18] W3C. SPARQL 1.1 Protocol. [online], Mar 2013, [visited on 2022-03-18]. Available from: https://www.w3.org/TR/sparql11-protocol/

[19] Foundation, O. About OpenStreetMap. [online], [visited on 2022-03-28]. Available from: https://www.openstreetmap.org/about

[20] LinkedGeoData. [online], [visited on 2022-03-29]. Available from: http://linkedgeodata.org/

[21] Registr územní identifikace, Adres a nemovitostí (RÚIAN). [online], 2022, [visited on 2022-04-01]. Available from: https://www.cuzk.cz/ruian/

[22] Wikidata. Dec 2019, [visited on 2022-04-01]. Available from: https://www.wikidata.org/wiki/Wikidata:Main_Page

[23] Sahoo, N.; Abdulhamid, A.; et al. Synchronising Wikidata and Wikipedia: An outreachy project. [online], Oct 2021, [visited on 2022-04-01]. Available from: https://diff.wikimedia.org/2021/10/01/synchronising-wikidata-and-wikipedia-an-outreachy-project/

[24] About Dbpedia. [online], Mar 2021, [visited on 2022-04-02]. Available from: https://www.dbpedia.org/about/

[25] JavaScript with syntax for types. [online], [visited on 2022-04-03]. Available from: https://www.typescriptlang.org/

[26] Turner, J. Announcing typescript 1.0. [online], Feb 2019, [visited on 2022-04-03]. Available from: https://devblogs.microsoft.com/typescript/announcing-typescript-1-0/

[27] RDF Triple Store FAQ. [online], [visited on 2022-04-04]. Available from: http://vos.openlinksw.com/owiki/wiki/VOS/VOSRDFFAQ

[28] W3C. Shapes Constraint Language (SHACL). [online], Jul 2017, [visited on 2022-04-04]. Available from: https://www.w3.org/TR/shacl/

[29] Sequeda, J. Introduction to: Open world assumption vs closed world assumption. [online], Jan 2015, [visited on 2022-04-04]. Available from: https://www.dataversity.net/introduction-to-open-world-assumption-vs-closed-world-assumption/

[30] Meta Platforms, I. React – a JavaScript library for building user interfaces. [online], [visited on 2022-04-01]. Available from: https://reactjs.org/

[31] Learning React Native. [online], 2020, [visited on 2022-04-05]. Available from: https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch02.html

[32] Inc, S. E. Stack overflow developer survey 2021. [online], [visited on 2022-04-05]. Available from: https://insights.stackoverflow.com/survey/2021

[33] NPM Trends - Angular vs React vs Vue. [online], [visited on 2022-04-05]. Available from: https://www.npmtrends.com/angular-vs-react-vs-vue

[34] Angular - the modern web developer's platform. [online], [visited on 2022-04-05]. Available from: https://angular.io/

[35] Vue.js - the progressive javascript framework. [online], [visited on 2022-04-05]. Available from: https://vuejs.org/

[36] Documentation: Nestjs - a progressive node.js framework. [online], [visited on 2022-04-06]. Available from: https://docs.nestjs.com/

[37] Documentation: Nestjs - a progressive node.js framework. [online], [visited on 2022-04-06]. Available from: https://docs.nestjs.com/modules

[38] Node.js web application framework. [online], [visited on 2022-04-06]. Available from: https://expressjs.com/

[39] Introduction. [online], [visited on 2022-04-06]. Available from: https://koajs.com/

[40] A fully featured web framework for Node.js. [online], [visited on 2022-04-06]. Available from: https://adonisjs.com/

[41] Ltd., R. Introduction to Redis. [visited on 2022-04-08]. Available from: https://redis.io/docs/about/

[42] OptimalBits. Bull's documentation. [online], [visited on 2022-04-08]. Available from: https://optimalbits.github.io/bull/

[43] Rachel Potvin, J. L. Why Google stores billions of lines of code in a single repository. [online], Jul 2016, [visited on 2022-04-09]. Available from: https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext

[44] Lerna - A tool for managing JavaScript projects with multiple packages. 2020, [visited on 2022-04-09]. Available from: https://lerna.js.org/

[45] Manning, C. D.; Raghavan, P.; et al. *Introduction to information retrieval*. Cambridge University Press, 2018.

[46] Fielding, R. T. Fielding dissertation: Chapter 5: Representational state transfer (rest). 2000, [visited on 2022-04-11]. Available from: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[47] Using round robin for simple load balancing. [online], Mar 2022, [visited on 2022-04-13]. Available from: https://www.nginx.com/resources/glossary/round-robin-load-balancing/

[48] Jest - delightful JavaScript testing. 2016, [visited on 2022-04-17]. Available from: https://jestjs.io/

[49] JavaScript end to end testing framework. [online], 2022, [visited on 2022-04-17]. Available from: https://www.cypress.io/

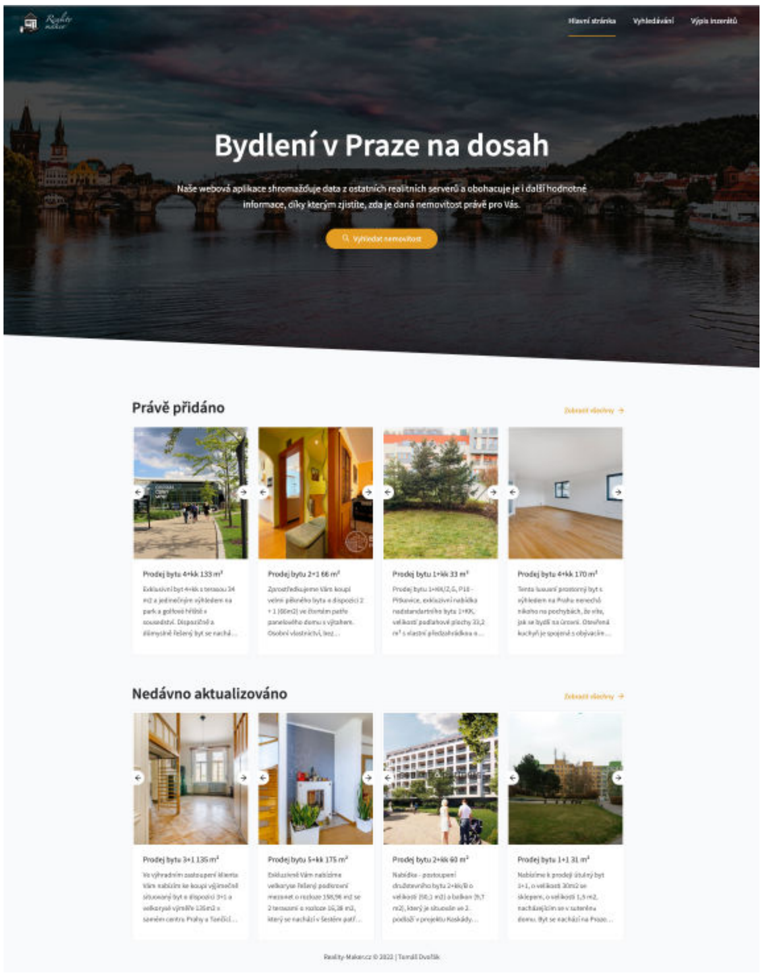
[50] Pittet, S. The different types of testing in software. [visited on 2022-04-17]. Available from: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

[51] Hotjar. What is usability testing? (and what it isn't). [online], Feb 2022, [visited on 2022-04-18]. Available from: https://www.hotjar.com/usability-testing/

[52] Nielsen, J. Why you only need to test with 5 users. [online], Mar 2000, [visited on 2022-04-18]. Available from: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/

Appendix A

Web application screenshots



Bydlení v Praze na dosah

Naše webová aplikace shromažďuje data z ostatních realitních serverů a obohacuje je i další hodnotné informace, díky kterým zjistíte, zda je daná nemovitost právě pro Vás.

Právě přidáno

Prodej bytu 4+kk 133 m²
Exklusivní byt 4+kk s terasou 34 m2 a jedinečným výhledem na park a golfové hřiště v sousedství. Dispozičně a důmyslně řešený byt se nachá…

Prodej bytu 2+1 66 m²
Zprostředkujeme Vám koupi velmi pěkného bytu o dispozici 2 + 1 (66m2) ve čtvrtém patře panelového domu s výtahem. Osobní vlastnictví, bez…

Prodej bytu 1+kk 33 m²
Prodej bytu 1+KK/Z,G, P18 - Pitkovice, exkluzivní nabídka nadstandartního bytu 1+KK, velikosti podlahové plochy 33,2 m² s vlastní předzahrádkou a…

Prodej bytu 4+kk 170 m²
Tento luxusní prostorný byt s výhledem na Prahu nenechá nikoho na pochybách, že víte, jak se bydlí na úrovni. Otevřená kuchyň je spojená s obývacím…

Nedávno aktualizováno

Prodej bytu 3+1 135 m²
Ve výhradním zastoupení klienta Vám nabízíme ke koupi výjimečně situovaný byt o dispozici 3+1 a velkorysé výměře 135m2 v samém centru Prahy a Tančící…

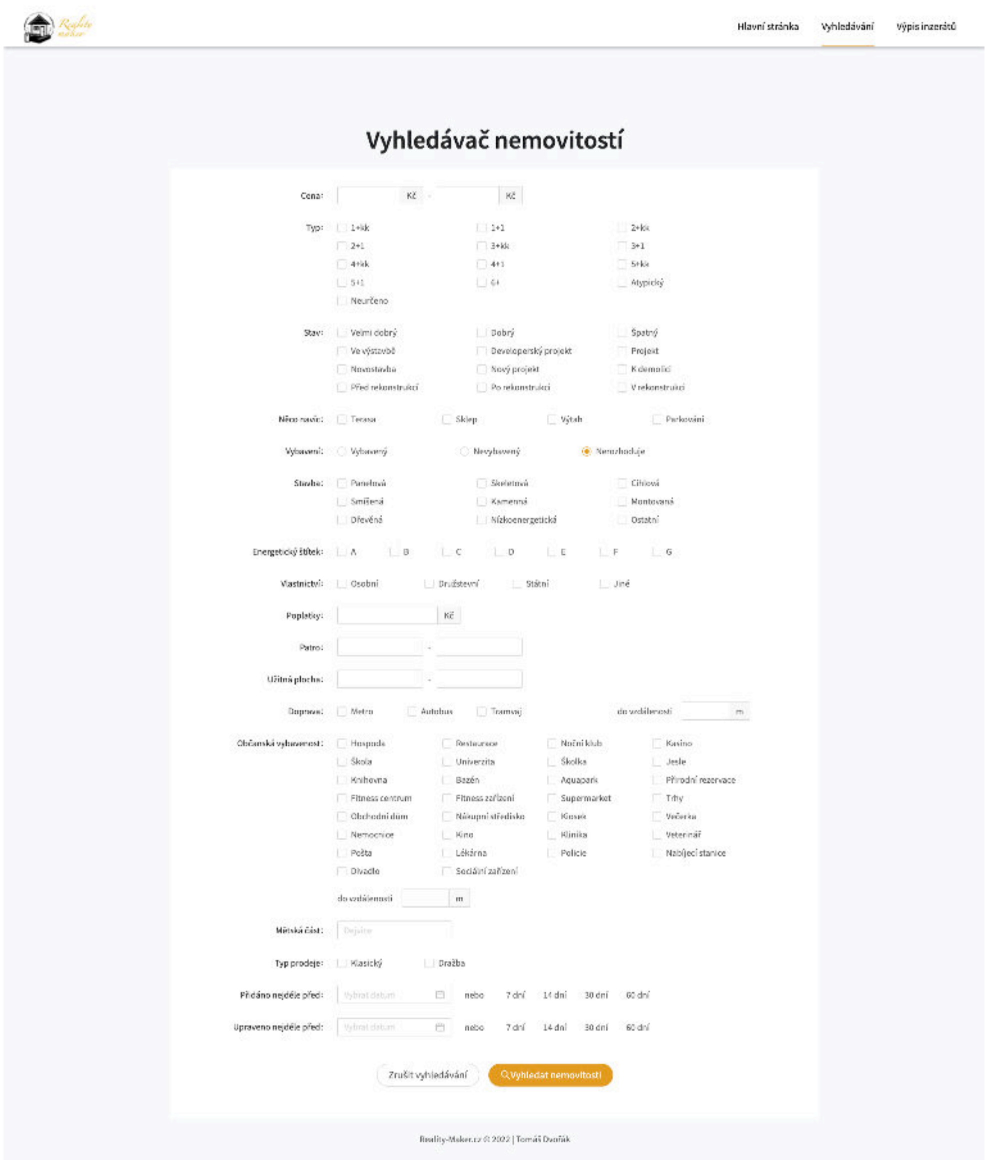
Prodej bytu 5+kk 175 m²
Exkluzivně Vám nabízíme velkoryse řešený podkrovní mezonet o rozloze 158,96 m2 se 2 terasami o rozloze 16,38 m2, který se nachází v šestém patř…

Prodej bytu 2+kk 60 m²
Nabídka - postoupení družstevního bytu 2+kk/B o velikosti (50,1 m2) a balkon (9,7 m2), který je situován ve 2. podlaží v projektu Kaskády…

Prodej bytu 1+1 31 m²
Nabízíme k prodeji útulný byt 1+1, o velikosti 30m2 se sklepem, o velikosti 1,5 m2, nacházejícím se v suterénu domu. Byt se nachází na Prazo…

Reality-Maker.cz © 2022 | Tomáš Dvořák

Figure A.1: A preview of the Homepage



Vyhledávač nemovitostí

Cena: Kč - Kč

Typ: 1+kk, 1+1, 2+kk, 2+1, 3+kk, 3+1, 4+kk, 4+1, 5+kk, 5+1, 6+, Atypický, Neurčeno

Stav: Velmi dobrý, Dobrý, Špatný, Ve výstavbě, Developerský projekt, Projekt, Novostavba, Nový projekt, K demolici, Před rekonstrukcí, Po rekonstrukci, V rekonstrukci

Něco navíc: Terasa, Sklep, Výtah, Parkování

Vybavení: Vybavený, Nevybavený, [✓ Nerozhoduje]

Stavba: Panelová, Skeletová, Cihlová, Smíšená, Kamenná, Montovaná, Dřevěná, Nízkoenergetická, Ostatní

Energetický štítek: A, B, C, D, E, F, G

Vlastnictví: Osobní, Družstevní, Státní, Jiné

Poplatky: Kč

Patro: -

Užitná plocha: -

Doprava: Metro, Autobus, Tramvaj, do vzdálenosti m

Občanská vybavenost: Hospoda, Restaurace, Noční klub, Kasino, Škola, Univerzita, Školka, Jesle, Knihovna, Bazén, Aquapark, Přírodní rezervace, Fitness centrum, Fitness zařízení, Supermarket, Trhy, Obchodní dům, Nákupní středisko, Kiosek, Večerka, Nemocnice, Kino, Klinika, Veterinář, Pošta, Lékárna, Policie, Nabíjecí stanice, Divadlo, Sociální zařízení

do vzdálenosti m

Městská část: Zadejte

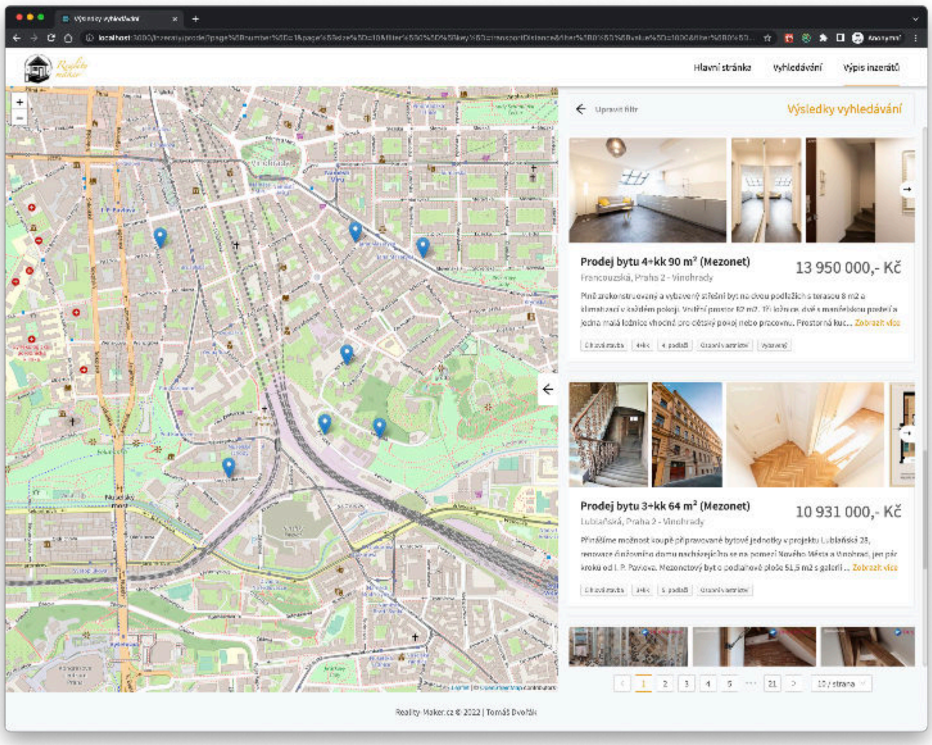
Typ prodeje: Klasický, Dražba

Přidáno nejdéle před: Vybrat datum nebo 7 dní 14 dní 30 dní 60 dní

Upraveno nejdéle před: Vybrat datum nebo 7 dní 14 dní 30 dní 60 dní

Reality-Maker.cz © 2022 | Tomáš Dvořák

Figure A.2: Preview of the Search page



Prodej bytu 4+kk 90 m^2 (Mezonet) 13 950 000,- Kč

Francouzská, Praha 2 - Vinohrady

Plně zrekonstruovaný a vybavený střešní byt na dvou podlažích s terasou 8 m2 a klimatizací v každém pokoji. Vnitřní prostor 82 m2. Tři ložnice, dvě s manželskou postelí a jedna malá ložnice vhodná pro dětský pokoj nebo pracovnu. Prostorná kuc... Zobrazit více

Cihlová stavba | 4+kk | 4. podlaží | Osobní vlastnictví | Vybavený

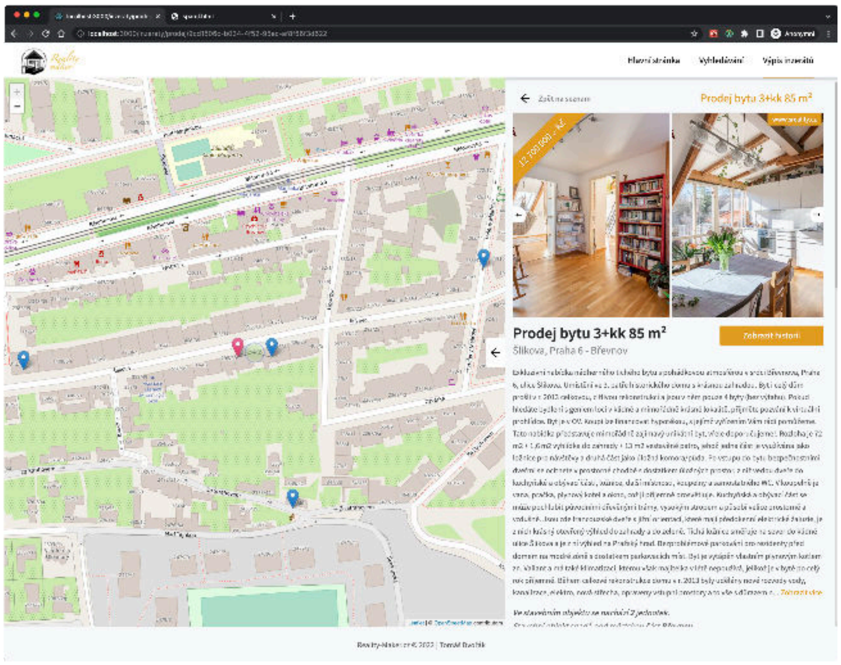
Prodej bytu 3+kk 64 m^2 (Mezonet) 10 931 000,- Kč

Lublaňská, Praha 2 - Vinohrady

Přinášíme možnost koupě připravované bytové jednotky v projektu Lublaňská 25, renovace činžovního domu nacházejícího se na pomezí Nového Města a Vinohrad, jen pár kroků od I. P. Pavlova. Mezonetový byt o podlahové ploše 51,5 m2 s galerií ... Zobrazit více

Cihlová stavba | 3+kk | 5. podlaží | Osobní vlastnictví

Figure A.3: Preview of the listing page depicting founded results



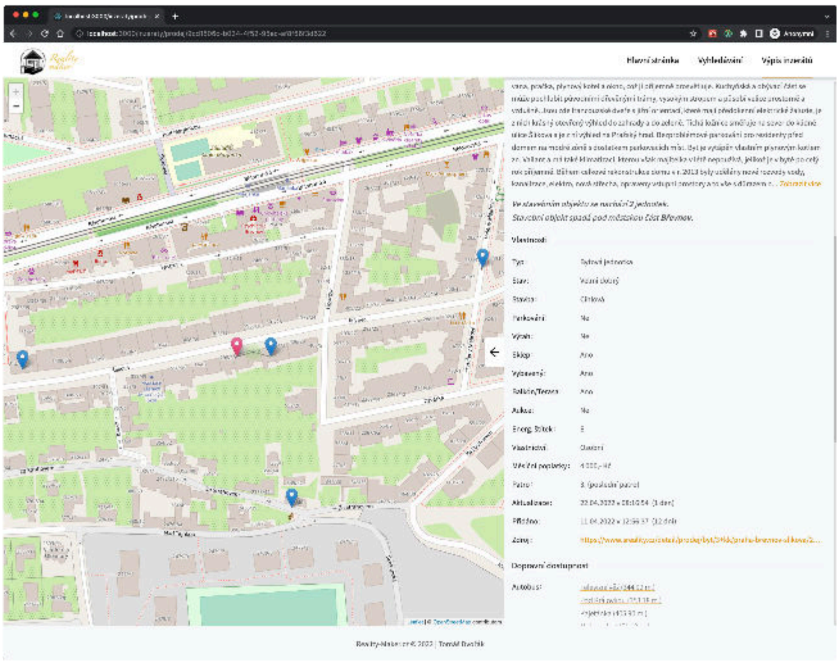
Prodej bytu 3+kk 85 m²

Šlikova, Praha 6 - Břevnov

[illegible]

Ve stavebním objektu se nachází 2 jednotek.

Figure A.4: Detail page of real estate (top view)



[illegible]

Ve stavebním objektu se nachází 2 jednotek.

Stavební objekt spadá pod městskou část Břevnov.

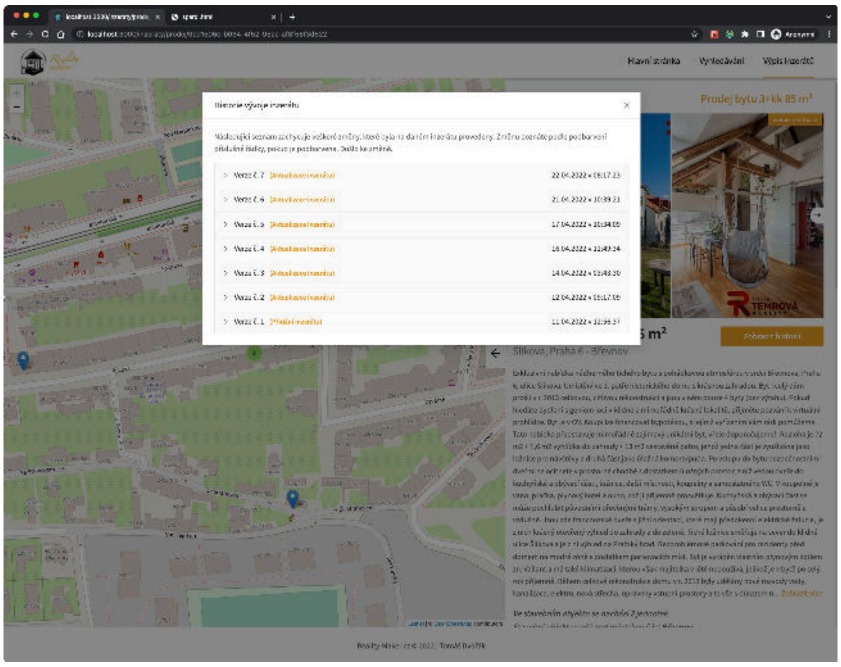
Vlastnosti

Typ:	Bytová jednotka
Stav:	Velmi dobrý
Stavba:	Cihlová
Parkování:	Ne
Výtah:	Ne
Sklep:	Ano
Vybavený:	Ano
Balkón/Terasa:	Ano
Aukce:	Ne
Energ. štítek:	E
Vlastnictví:	Osobní
Měsíční poplatky:	4 000,- Kč
Patro:	3. (poslední patro)
Aktualizace:	22.04.2022 v 08:10:54 (1 den)
Přidáno:	11.04.2022 v 12:56:37 (12 dnů)
Zdroj:	https://www.reality...

Dopravní dostupnost

Autobus: [illegible]

Figure A.5: Detail page of real estate (bottom view)

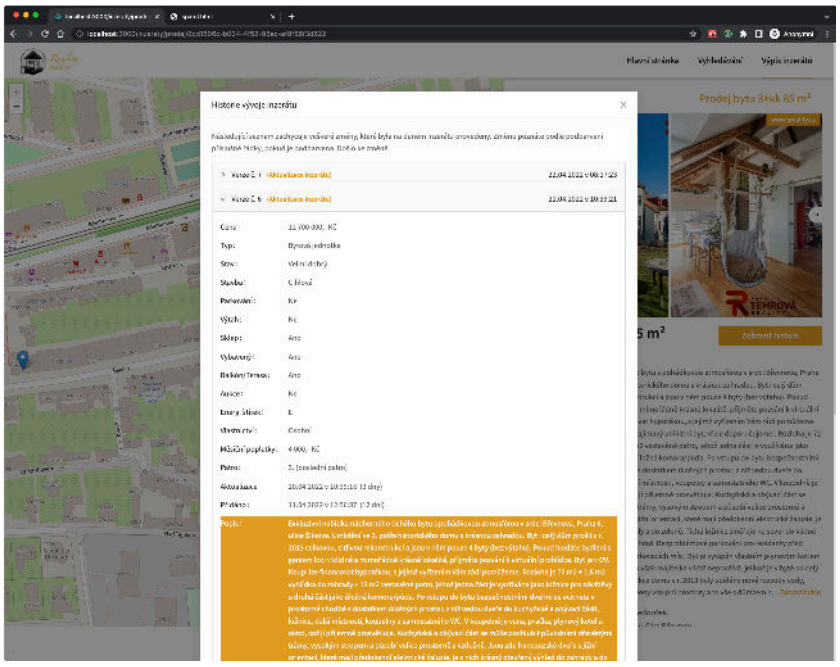


Historie vývoje inzerátu

Následující seznam zachycuje veškeré změny, které byly na daném inzerátu provedeny. Změnu poznáte podle podbarvení příslušné řádky, pokud je podbarvena. Došlo ke změně.

> Verze č. 7	(Aktualizace inzerátu)	22.04.2022 v 08:17:23
> Verze č. 6	(Aktualizace inzerátu)	21.04.2022 v 10:39:21
> Verze č. 5	(Aktualizace inzerátu)	17.04.2022 v 10:34:05
> Verze č. 4	(Aktualizace inzerátu)	16.04.2022 v 12:40:24
> Verze č. 3	(Aktualizace inzerátu)	14.04.2022 v 13:48:30
> Verze č. 2	(Aktualizace inzerátu)	12.04.2022 v 05:17:06
> Verze č. 1	(Přidání inzerátu)	11.04.2022 v 12:56:37

Figure A.6: Collapsed view of a real estate history



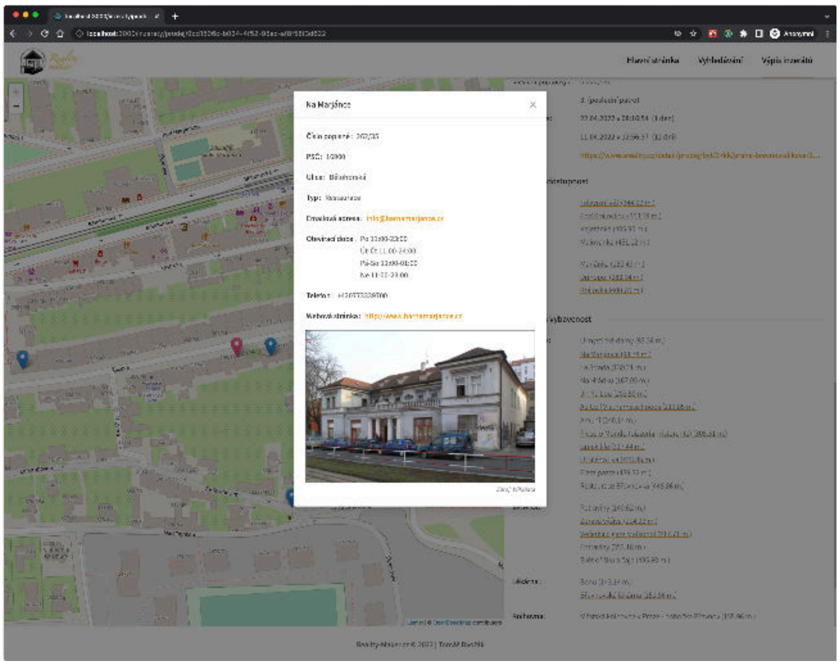
Historie vývoje inzerátu

Následující seznam zachycuje veškeré změny, které byly na daném inzerátu provedeny. Změnu poznáte podle podbarvení příslušné řádky, pokud je podbarvena. Došlo ke změně.

> Verze č. 7	(Aktualizace inzerátu)	22.04.2022 v 08:17:23
˅ Verze č. 6	(Aktualizace inzerátu)	21.04.2022 v 10:39:21

Cena:	12 700 000,- Kč
Typ:	Bytová jednotka
Stav:	Velmi dobrý
Stavba:	Cihlová
Parkování:	Ne
Výtah:	Ne
Sklep:	Ano
Vybavený:	Ano
Balkón/Terasa:	Ano
Aukce:	Ne
Energ. štítek:	E
Vlastnictví:	Osobní
Měsíční poplatky:	4 000,- Kč
Patro:	3. (poslední patro)
Aktualizace:	20.04.2022 v 10:35:16 (1 dny)
Přidáno:	11.04.2022 v 12:56:37 (12 dní)
Popis:	Exkluzivní nabídka nádherného tichého bytu s pohádkovou atmosférou v srdci Břevnova, Praha 6, ulice Šlikova. Umístění ve 3. patře historického domu s krásnou zahradou. Byt i celý dům prošli v r. 2013 celkovou, citlivou rekonstrukcí a jsou v něm pouze 4 byty (bez výtahu). [illegible]

Figure A.7: Expanded view of a real estate history


Na Marjánce

Číslo popisné: 262/35

PSČ: 16900

Ulice: Bělohorská

Typ: Restaurace

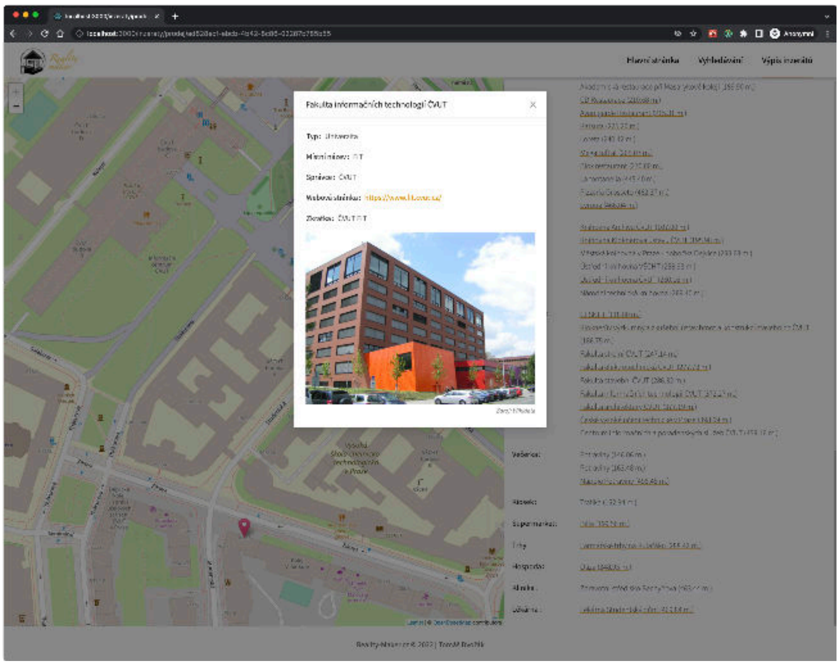
Emailová adresa: info@barnamarjance.cz

Otevírací doba: Po 11:00-23:00
Út-Čt 11:00-24:00
Pá-So 11:00-01:00
Ne 11:00-23:00

Telefon: +420777009700

Webová stránka: http://www.barnamarjance.cz

Figure A.8: Place detail (OSM & Wikidata)


Fakulta informačních technologií ČVUT

Typ: Univerzita

Místní název: FIT

Správce: ČVUT

Webová stránka: https://www.fit.cvut.cz/

Zkratka: ČVUT FIT

Figure A.9: Place detail (OSM & Wikidata)

Appendix B

List of Acronyms

API Application Programming Interface. xiii, 12, 23, 28–31, 33, 35, 44, 45, 55, 57, 60, 63, 64, 66, 67, 70, 78

CD Continuous Delivery. 32

CI Continuous Integration. 32

CLI Command Line Interface. 47

CRUD Create-Read-Update-Delete. 49, 53

CSR Client Side Rendering. 5

CSS Cascading Style Sheets. 55, 57, 60

CSV Comma-separated values. 18

DOM Document Object Model. 26, 27, 55, 57

DRY Do not Repeat Yourself. 31

DTO Data Transfer Object. 29, 67, 68

E2E End-to-End. xvi, 31, 75, 76, 78–80

GIT Global Information Tracker. 31

HTML Hypertext Markup Language. 5, 13, 15, 19, 26, 27, 50, 55, 57

HTTP Hypertext Transfer Protocol. 14, 15, 17, 25, 26, 28, 31, 67, 68

IPC Inter Process Communication. 50

IRI Internationalized Resource Identifier. 13, 14, 52

JDBC Java DataBase Connectivity. 25, 49, 50

JSON JavaScript Object Notation. 44, 57, 67

JSX JavaScript Syntax Extension. 27

JVM Java Virtual Machine. 50

MVC Model-View-Controller. 28, 30, 67

NLP Natural Language Processing. 37, 57

NPM Node Package Manager. 31

OAS Open API Specification. 67

OCP Open Closed Principle. 56

OOP Object Oriented Programming. 56, 75

OSM OpenStreetMap. xiv, xv, 17, 18, 20, 45, 64, 66, 72, 74

OWL Web Ontology Language. xvi, 13, 15, 111

PDB Protein Data Bank. 18

POI Point of Interest. 44, 45

RDBMS Relational Database Management System. 25

RDF Resource Description Framework. xv, xvi, 13–15, 17, 18, 20, 25, 26, 52, 53, 76, 112

RDFS Resource Description Framework Schema. xvi, 13–15, 111

REST Representational State Transfer. 35, 44

RUIÁN Register of Territorial Identification, Addresses and Real Estate. xv, 18, 20, 64, 74

SEO Search Engine Optimalization. 60

SHACL Shapes Constraint Language. 26

SPA Single Page Application. 12, 78

SPARQL Simple Protocol and RDF Query Language. xv, 13, 15–20, 25, 49, 51, 53, 54, 70–74, 77

SQL Structured Query Language. 16, 25, 26

UI User Interface. 21, 31, 60, 61, 78

URI Uniform Resource Identifier. 15

URL Uniform Resource Locator. 7, 19, 35, 36, 56, 60, 61, 63, 78, 79

VDOM Virtual Document Object Model. 26, 27

XML Extensible Markup Language. 17, 18

YAML YAML Ain't Markup Language. 67

Appendix C

List of Source codes

```
@Injectable()
export class RealEstateHistorySubscriber implements
  IEntitySubscriber<RealEstateEntity> {

  protected readonly rdfProperties =
    EntityTransformer.getRdfProperties(RealEstateEntity)

  // Passing required parameters is done dependency injection mechanism
  constructor(
    protected readonly entityRepository: RealEstateRepository,
    protected readonly historyRepository: RealEstateHistoryRepository
  ) {
    // public methods will be called when occurs once the changes
    // occurs on RealEstateEntity
    this.startListening()
  }

  async beforeDelete(newPartialEntity: Partial<RealEstateEntity>) {
    const id = newPartialEntity.id
    const oldEntity = await this.entityRepository.findById(id)

    await this.saveChanges(
      oldEntity,
      newPartialEntity,
      [], // no changed fields
      SharedConstant.HistoryOperation.DELETED
    )
  }

  async beforeUpdate(newPartialEntity: Partial<RealEstateEntity>) {
    const id = newPartialEntity.id
    const oldEntity = await this.entityRepository.findById(id)

    const newEntity: RealEstateEntity =
      await transformAndValidate(RealEstateEntity, {
        ...oldEntity,
        ...newPartialEntity,
      })

    const changedFields: NamedNode[] =
      Object.keys(diffEntities(newEntity, oldEntity!))
        .filter((key) => this.rdfProperties.hasOwnProperty(key))
        .map((key) => this.rdfProperties[key].node)

    if (changedFields.length > 0) {
      await this.saveChanges(
        oldEntity,
        newEntity,
        changedFields,
        SharedConstant.HistoryOperation.UPDATED
      )
    }
  }

  async beforeInsert(entity: RealEstateEntity) {
    const historyEntityPlain: IRealEstateHistory = {
      ...entity,
      changedField: [],
      version: 1,
      operation: SharedConstant.HistoryOperation.CREATED,
    }

    const classInstance = await transformAndValidate(
      RealEstateHistoryEntity,
      historyEntityPlain
    )

    await this.historyRepository.insert(classEntity)
  }

  startListening() {
    this.entityRepository.subscribe(this)
  }

  stopListening() {
    this.entityRepository.unsubscribe(this)
  }

  // Creates new history snapshot
  protected async saveChanges(
    oldEntity: RealEstateEntity,
    newEntity: RealEstateEntity,
    changedField: NamedNode[],
    operation: SharedConstant.HistoryOperationType
  ) {
    const lastEntity = await this.historyRepository.findOne({
      select: ['version'],
      where: [
        {
          key: 'source',
          value: newEntity.source,
          operator: '=',
        },
      ],
      order: [
        {
          key: 'version',
          desc: true,
        },
      ],
    })

    const version = lastEntity ? lastEntity.version + 1 : 1
    const plainHistoryEntity: IRealEstateHistory = {
      ...oldEntity,
      ...newEntity,
      changedField,
      operation,
      version,
    }

    const historyInstance = await transformAndValidate(
      RealEstateHistoryEntity,
      plainHistoryEntity
    )
    await this.historyRepository.insert(historyInstance)
  }
 }
```

Listing C.1: Implementation of *RealEstateHistorySubscriber*

```
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .
@prefix interval:  <http://reference.data.gov.uk/def/intervals/> .
@prefix qb:  <http://purl.org/linked-data/cube#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix euvoc: <http://publications.europa.eu/ontology/euvoc#> .
@prefix realm: <http://reality-maker.cz/vocabulary/> .
@prefix dcterms: <http://purl.org/dc/terms/> .

# Class definitions
realm:RealEstate rdf:type rdfs:Class, owl:Class ;
  rdfs:label "Real Estate" ;
  rdfs:comment "Sell advertisement for Real Estate - flat"@en ;
  rdfs:isDefinedBy realm: .

realm:RealEstateHistory rdf:type rdfs:Class, owl:Class ;
  rdfs:label "Real Estate History" ;
  rdfs:comment "Snapshot of RealEstate entity in given time."@en ;
  rdfs:subClassOf realm:RealEstate ;
  rdfs:isDefinedBy realm: .

# Property definitions
realm:version rdf:type rdf:Property, owl:FunctionalProperty ;
```

```
  rdfs:domain realm:RealEstateHistory ;
  rdfs:range xsd:positiveInteger ;
  rdfs:comment "Number of current version."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:changedFields rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstateHistory ;
  rdfs:range [ owl:oneOf ("CREATED" "UPDATED" "DELETED") ] ;
  rdfs:comment "Reason why entry was created"@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:operation rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstateHistory ;
  rdfs:range xsd:anyURI ;
  rdfs:comment "IRI to a property which has changed"@en ;
  owl:minCardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:images rdf:type rdf:Property, owl:ObjectProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:anyURI ;
  rdfs:comment "Related image"@en ;
  rdfs:isDefinedBy realm: .

realm:annuity rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:nonNegativeInteger ;
  rdfs:comment "Remaining annuity (payment) in CZK currency."@en ;
  owl:maxCardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:energyLevel rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range [ owl:oneOf ("A" "B" "C" "D" "E" "F" "G") ] ;
  rdfs:comment "Energy level of electric consumption
    (A - best, G - worst). "@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:floor rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:integer ;
  rdfs:comment "Floor where the given flat belongs to.
    Zero equals to ground floor."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

```

```
realm:usableArea rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:positiveInteger ;
  rdfs:comment "Fully-enclosed space that is available for
    the exclusive use of a building
      occupant for personnel, materials, furniture,
      fixtures and equipment. Exluding internal partitions."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:floorArea rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:positiveInteger ;
  rdfs:comment "Sum of whole are of flat space
    including internal partitions."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:layout rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range [ owl:oneOf ("1+kk" "1+1" "2+1" "3+kk" "3+1" "4+1"
    "4+kk" "5+1" "5+kk" "6+1" "atypic" "unknown" ) ] ;
  rdfs:comment "Layout of the given estate."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:ownership rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:string ;
  rdfs:comment "Type of legal entity who owns the estate."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:structure rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:string ;
  rdfs:comment "Material with building is made of"@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:state rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:string ;
  rdfs:comment "State of the given estate."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:unitType rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:string ;
  rdfs:comment "Law status describing
     the usage of the given property."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:monthlyFees rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:nonNegativeInteger ;
  rdfs:comment "Monthly fees in CZK currency."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:price rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:nonNegativeInteger ;
  rdfs:comment "Price of the Real estate in CZK currency."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:hasCellar rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:boolean ;
  rdfs:comment "Tels if given estate has a cellar."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:hasLift rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:boolean ;
  rdfs:comment "Tels if given estate has a cellar."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:hasParking rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:boolean ;
  rdfs:comment "Tels if given estate has a parking place."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:hasTerrace rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:boolean ;
  rdfs:comment "Tels if given estate has a terrace."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:isAuction rdf:type rdf:Property, owl:FunctionalProperty ;
```

```
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:boolean ;
  rdfs:comment "Tels if the given real estate
    advertisment is an auction."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:isLastFloor rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:boolean ;
  rdfs:comment "Tels if the given flat is in the last floor."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:withEquipment rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:range xsd:boolean ;
  rdfs:comment "Tells if the real estate is with equipment."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .

realm:sourceUpdatedAt rdf:type rdf:Property, owl:FunctionalProperty ;
  rdfs:domain realm:RealEstate ;
  rdfs:subPropertyOf dcterms:modified ;
  rdfs:range xsd:dateTime ;
  rdfs:comment "Reflects the date time
    when the source has been updated."@en ;
  owl:cardinality "1"^^xsd:nonNegativeInteger ;
  rdfs:isDefinedBy realm: .
```

Listing C.2: Defined vocabulary for the project needs in RDFS/OWL

```
@prefix realm: <http://reality-maker.cz/vocabulary/>.
@prefix entity: <http://reality-maker.cz/entity/> .
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix schema: <http://schema.org/>.
@prefix wgs: <http://www.w3.org/2003/01/geo/wgs84_pos#>.
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

entity:2ecf75f5 a realm:RealEstate;
    dcterms:title "Prodej bytu 3+kk 36 m"@cs;
    dcterms:description "Prodej bytu 3+kk s výměrou..."@cs;
    dcterms:created "2022-03-27T20:00:40.507Z"^^xsd:dateTime;
    dcterms:modified "2022-03-28T22:02:46.507Z"^^xsd:dateTime;
    dcterms:identifier "2ecf75f5-2ba2-48e9-8afa-7185189016ec";
    realm:sourceModified "2022-03-27T22:02:41.983Z"^^xsd:dateTime.
    realm:source <https://www.sreality.cz/detail/prodej/123>;
    wgs:long "14.4371415"^^xsd:double;
    wgs:lat "50.0173345"^^xsd:double;
    schema:address "Zvolská, Praha 4 - Kamýk";
    realm:withEquipment "true"^^xsd:boolean;
    realm:isLastFloor "false"^^xsd:boolean;
    realm:isAuction "false"^^xsd:boolean;
    realm:hasTerrace "true"^^xsd:boolean;
    realm:hasParking "false"^^xsd:boolean;
    realm:hasLift "true"^^xsd:boolean;
    realm:hasCellar "true"^^xsd:boolean;
    realm:unitType "Bytová jednotka"@cs;
    realm:structure "Panelová"@cs;
    realm:price "4990000"^^xsd:nonNegativeInteger;
    realm:ownership "Osobní"@cs;
    realm:monthlyFees "3586"^^xsd:nonNegativeInteger;
    realm:state "Velmi dobrý"@cs;
    realm:layout "3+kk";
    realm:usableArea "36"^^xsd:positiveInteger;
    realm:floorArea "36"^^xsd:positiveInteger;
    realm:floor "4"^^xsd:integer;
    realm:energyLevel "G".
```

Listing C.3: The real estate entity described in RDF in *Turtle* serialization

Appendix **D**

Content of enclosed SDHC Card

- `README.md` .... description of the contents of the sub-directories
- `src` .... the directory of source codes
  - `README.md` .... description with installation steps
  - `apps` .... the directory with runnable application
    - `api` .... REST API
    - `web` .... Web Application
    - `scraper` .... Scraper Service
    - `analyser` .... Analyser Service
  - `libs` .... the directory with shared libraries
    - `core`
    - `shared`
- `thesis.pdf` .... the thesis text in PDF format
- `latex` .... the directory of LaTeXsource codes of the thesis