#MetaFood
- One App To Feed Them All -

Final Report for
Semantic Systems VU
TU Wien, 188.399, WS 2019/20

Group 30
Amer Alkojjeh, 01646631 - Business Informatics (E 066 926)
Thassilo Gadermaier, 00507477 - Data Science (E 066 645)
Philip Klaus, 01407087 - Software Engineering (E 066 937)

February 20, 2020

Contents

Introduction 4

1. Task 1: The Application 4
1.1. Storytelling the app: hangry at the office . . . 4
1.2. Description . . . 4
1.3. Competency Questions . . . 5
1.4. Images and UI mockups . . . 5

2. Task 2: Data Collection 5
2.1. General workflow . . . 6
2.1.1. Getting raw data . . . 6
2.1.2. Workflow for cleaning and processing . . . 7
2.1.3. Workflow for merging . . . 7
2.1.4. Data cleanup . . . 7
2.2. Uber eats . . . 8
2.2.1. Collection process . . . 8
2.2.2. Raw data description . . . 8
2.3. Supper . . . 10
2.3.1. Collection process . . . 10
2.3.2. Raw data description . . . 10
2.4. Foodhub . . . 11
2.4.1. Collection process . . . 11
2.4.2. Raw data description . . . 11

3. Task 3: Creating the Ontology 12
3.1. External Ontologies . . . 12
3.2. Building Process . . . 13

4. Task 4: Creating a Knowledge Graph 15

5. Task 5: Knowledge Graph to Triple Store 17

6. Task 6: SPARQL Queries 19

7. Task 7: Creating the Application 21
7.1. User interface . . . 21
7.2. Technological details . . . 23
7.2.1. Workflow . . . 23
7.2.2. The webserver . . . 24
7.2.3. The Graph Database . . . 24
7.2.4. Running the application . . . 24

A. Additional Data **24**
A.1. Deliveroo 25
A.1.1. Collection process 25
A.1.2. Raw data description 25

Introduction

This report documents our efforts for the project for practical part of the class.

The report's structure is based on the tasks given in the assignment. First is a description of the application, describing the idea, and a formulation of a few competency questions that it can answer. Next, we describe the process of acquiring the necessary data, and a bit of how we organized them. The following chapters the document creation of the ontology and knowledge graph, and transition to a triple store. After showing some relevant SPARQL queries, we will give an overview of the Webapp built around the system.

We would like to mention that this group started as a 4 piece. One member decided to quit the project (and class) after some time. He contributed to the data collection process by producing code for loading and also acquiring one data set, and documented his efforts. The data were not further processed by us, as we seemed to have available enough data from the 3 remaining sources. We decided ultimately to keep the data in the submission, as well as the documentation, which can be found in the Appendix A.

1. Task 1: The Application

1.1. Storytelling the app: hangry at the office

At the office, Jenny, Tom and Anna are hangry (hungry + angry). It's time for lunch, and they again can't agree on where to go for lunch. Mike has an idea: he discovered this new app, called MetaFood, that allows for better organizing orders from food delivery services. Mike passes around his phone, and everybody enters his or her favorite food. The app suggests the highest rated restaurants in the vicinity that can fulfill the respective orders. After all orders are placed, the system in the backend will automatically coordinate the orders in time, based on typical delivery times to the office, such that they will arrive at almost the same time. Jenny, Tom, Anna and Mike enjoy their lunch together.

1.2. Description

The main feature is that it allows for simply stating each persons favorite dishes, and a desired delivery time. The system will automatically, based on the location, time of order, desired delivery, etc., look for restaurants, that can fulfill the order in time. But not only any restaurant, but the ones with the highest ratings, availability, or your favorite, that you and your friends most often ordered from. The system coordinates the ordering processes to the different restaurants, as well as the delivery companies, to ensure that the food deliveries will converge at the given location (address) within a certain time frame. The system can give feedback to the individual deliverers for easier coordination of convergence.

1.3. Competency Questions

In the following we present some competency questions (CQ). During the ideation and conception phase, we came up with a larger number if CQ, but during further development of our system, we then aligned them with the envisaged functionality and narrowed them down to the ones presented here:

- CQ 1: Which delivery service delivers food X in less than 30 minutes?
- CQ 2: Which delivery service delivers free drinks with Food X?
- CQ 3: With which delivery service can I order food from the X highest rated restaurants in district Y?
- CQ 4: From which delivery service can I order food X with price less than Y pounds?
- CQ 5: Which delivery service delivers to postal code X and allows to pay by credit card/cash?
- CQ 6: which delivery service delivers the highest rated pizza Y to postal code X?
- CQ 7: which delivery service delivers gluten-free/lactose-free/halal/kosher food to postal code X?
- CQ 8: Which delivery service delivers food X and drink Y with the cheapest price in district Z?
- CQ 9: Which delivery service delivers high rated food X to district Y and accepts cash as payment method?

1.4. Images and UI mockups

During the design phase of our app, we came up with the story told above, and produced some draft sketches for the App's user interface (UI), as well as for how the ordering process and coordination is supposed to work. Figure 1 shows these sketches.

The documentation and description of the actual App produced is given in Section 7.

2. Task 2: Data Collection

This section describes the process of identifying appropriate possible sources; first experiments with getting data from them, exploration whether the data are suitable for envisaged application; settling on data sources and getting, cleaning and merging data from four different sources. We implemented all functionality for getting and processing the data in python. The actual organization at this stage happened using pandas dataframes.

It seemed natural to look first for data sources of delivery companies covering Vienna, however, due to the high market pressure and the size of the city, there are not enough

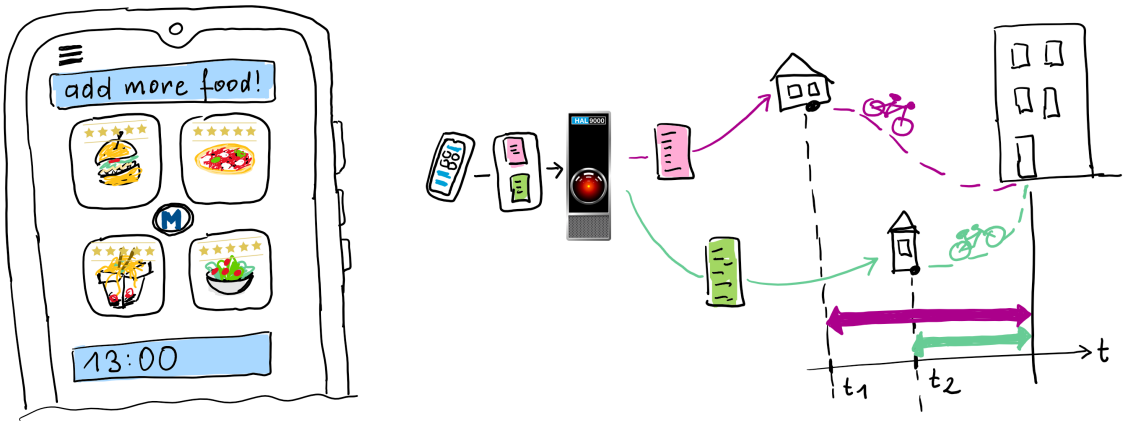

Figure 1: Sketches for the App's UI (left), and an overview of the system (backend) (right). The system splits orders placed at the same time into separate orders for each respective restaurant and coordinates the delivery in time such that they should arrive at the same target delivery time. See text for details.

delivery companies active (anymore). So, after some research we settled on the city of London (UK), where plenty of companies offer delivery services. More concretely, we selected the following four companies (and websites, respectively):

1. Uber Eats (`https://ubereats.com`)
2. Supper (`https://supper.london`)
3. Foodhub (`https://foodhub.co.uk`)
4. Deliveroo (`https://deliveroo.co.uk`)

While each data source was assigned to be (mainly, but not exclusively) handled and processed by a single team-member each, we communicated and shared our experiences quite tightly between each other. The respective main responsible team-member will be stated for each source.

2.1. General workflow

2.1.1. Getting raw data

The data we used for the project were publicly available (from the websites mentioned above, and more detailed in the dedicated sections below). As no personal data were required, we had no used for requests following GDPR.

While we used similar methods for getting the data from the different sources, we decided for organizing the code for capturing the pure data in one dedicated file (more concretely: one Jupyter-notebook) per source, even if this meant some copy-pasting of the re-used code. We found this easier for a parallel workflow of the team-members, especially as we experienced merging different versions of Jupyter-notebooks using git somewhat cumbersome.

The raw data consists in each case of the return from an HTTP request, stored as a dictionary in Python. For finding suitable portions of such a convoluted dump, visual inspection of the dictionary was done, to learn the structure, and find the relevant keys for extracting the necessary parts. As mentioned above [check?], we here already learned about useful attributes of the data. Extracted attributes' data were finally organized in a Pandas dataframe, where we pre-defined the column names (=attributes) beforehand for all four sources, after having identified all (at this stage) relevant attributes.

2.1.2. Workflow for cleaning and processing

As work on the data from the different sources was conducted mainly in parallel for two sources at first, we early on got an understanding of the different structures the data will come in, and the need for mapping slightly different attributes to each other, and thus for setting up a common structure for the data. After looking at some data, we defined a list of attributes, including suitable data types, that would cover the relevant attributes. The definitions for the attributes and their types, for data on restaurants, was well as on single menu items, were stored in the file `common_definitions.py`. For each source, a mapping from the present attributes to the ones in the common data structure was set up, such that combining the data from different sources would be easy. This somewhat anticipated the design of our ontology, and in fact we ended up going back and forth between processing the data (defining meaningful attributes) and designing the ontology, for some time.

2.1.3. Workflow for merging

The goal here was to collect all data from the different sources into one dataframe (and file), that should be easily convertible to final format. In fact, as described above, we had early on taken measures for easier merging of the data from different sources. Thus, merging of different sources amounted only to concatenating the Pandas dataframes of the different sources. The relevant code can be found in the file `merge_data_sources.ipynb`. The files with the merged data from (3 sources) can be found in the file `merged_data.json` in the folder `Task2_Getting_Data/Data/`.

2.1.4. Data cleanup

Data cleanup was conducted after the merging of the data from the different sources (but could have been done mostly at an earlier stage). Because we hat a lot of entries (about 8700 foods/menu items after raw data extraction) across the different restaurants and data sources, we removed all entries where no address or price was present, since we considered these as very central attributes for our data. After this initial removal, we had slightly more than 2500 entries left.

2.2. Uber eats

(This source was mainly processed by Thassilo Gadermaier).
Relevant code, within folder `Task2_Getting_Data`:

- `Uber_eats_getting_data.ipynb` is the script used for collecting the data.
- `response_victoria_street.json` is the raw data file for this source; the folder Uber_eats_Details contains further raw data (menus of restaurants).
- `/Data/uber_eats_with_menu.json` is the raw data file for this source.

For each source presented in the following, there may be additional files present, not mentioned here. These files are additional, auxiliary files created during the process of getting the data, and are not directly relevant.

2.2.1. Collection process

Uber offers an API for registered customers. Since we only needed some data, we decided for simply trying to get data about restaurants without any formal requests. Thus, we simply searched for restaurants at a certain location (Address) in London directly on the webpage `www.ubereats.com/en-GB/london`. As address for getting data, we selected Victoria Street in the district Westminster. After copying the HTTP request string, we experimented for a bit in Python using the `requests` library. This allows for directly forming HTTP requests from Python, and getting the response data. The response data was directly saved to JSON file on disk.

The response data was then processed such that for each restaurant found in the list, the relevant attributes, such as name, address, etc., were extracted and put into a dedicated dataframe. For getting each restaurant's menu (i.e. a list of dishes), a dedicated HTTP request per restaurant is formulated and executed for obtaining the menu data, which are dumped to auxiliary files. Often, the response for the menu data contained extra info on the respective restaurant, and such the restaurant data were enriched.

2.2.2. Raw data description

To give an overview of the raw data, we show an entry for a single restaurant. Some relevant attributes can be spotted, such as `'MinimumOrder'`.

Listing 1: Restaurant information in JSON

```
1
2 {'Description': 'Mayfair',
3  'Notes': '<h5 class="title">An eclectic, imaginative menu draws on
      Chef Jean-Georges Vongerichten\'s love of the city</h5> <h5 class
      ="title">and his experiences in the Far East.</h5> <h5 class="
      title">Beloved British and Connaught classics stand alongside
      South-east Asian flavours, crafted from</h5> <h5 class="title">
```

```
    fresh-from-the-market ingredients and farm-to-table produce.</h5
    >',
 'Recommendations': None,
 'ToppingsSets': None,
 'Locations': None,
 'OpeningHours': None,
 'IsOpened': True,
 'MinimumOrder': 20.0,
 'DeliveryFee': 6.25,
 'ServiceCharge': 0.0,
 'RestaurantCommision': 25.0,
 'IsActive': True,
 'WaitingTime': 0,
 'OrderNo': 0,
 'MobileDescription': None,
 'UpsellsSetId': 0,
 'UpsellsSet': None,
 'AdvancedOrdersDelay': False,
 'Notifications': None,
 'DeliveryPriceRanges': None,
 'ChangeTime': '2020-01-03T10:43:32+00:00',
 'ChangedByUser': '0551f96c-c7c1-476d-bd20-c47c5c04cda5',
 'FoodPreparationTime': None,
 'LocationId': None,
 'LocationIsOpened': True,
 'ConcatName': None,
 'LocationDeliveryRange': None,
 'LocationFoodPreparationTimeWorkload': None,
 'LocationPhoneNumber': None,
 'RestaurantId': 'e7ed2465-17df-4421-8301-ed13754a05d4',
 'Cuisines': [{'CuisineId': 22, 'CuisineName': 'French'},
  {'CuisineId': 24, 'CuisineName': 'British'},
  {'CuisineId': 31, 'CuisineName': 'Wine'},
  {'CuisineId': 33, 'CuisineName': 'Sweet treats'}],
 'RestaurantCode': 'jeangeorgesattheconnaught',
 'RestaurantName': 'Jean-Georges at The Connaught',
 'LogoImageUrl': 'jeangeorgesattheconnaught/jeangeorgesattheconnaught-
    logo.jpg',
 'MainImageUrl': 'jeangeorgesattheconnaught/jeangeorgesattheconnaught-
    main.jpg',
 'WebsiteUrl': 'https://www.the-connaught.co.uk/restaurants-bars/jean-
    georges-at-the-connaught/',
 'OtherImageUrls': [],
 'MobileImages': None}
```

An example for a menu item:

Listing 2: Menu item information in JSON.

```
{'description': 'Two poached eggs on Greek yoghurt, topped with spiced
   garlic butter and fresh dill and dukka, and with sourdough toast.',
 'imageUrl': '',
 'price': 975,
 'title': 'Turkish Eggs',
```

```
 'uuid': '0d61b818-9d49-40e9-b778-44d1330a12cf',
 'nutritionalInfo': {'allergens': [''], 'displayString': ''},
 'suspendReason': None,
 'suspendUntil': 0,
 'endorsement': {'text': 'Vegan',
  'iconUrl': 'https://uber-test.s3.amazonaws.com/badge_healthy@3x.png
     '},
 'classifications': []}
```

2.3. Supper

(This source was mainly processed by Philip Klaus).
Relevant code, within folder Task2_Getting_Data:

- `Supper_London_getting_data.ipynb` is the script used for collecting the data.
- `Supper_London_response_victoria_street.json` is the raw data file for this source; the folder Supper_London_Details contains further raw data (menus of restaurants).
- `/Data/supper_london_with_menu.json` is the processed data file for this source, where info on restaurants and menu data were merged.

2.3.1. Collection process

A similar procedure as described above was carried out for this source. For collecting the data for the same region, we used the post code "SW1H 0NF" for Westminster as parameter for the HTTP request for getting a list of restaurants. Again, lots of visual inspection of the response data was necessary to find the relevant attributes, and map their names (keys into the dictionary) to our common data structure's attributes.

2.3.2. Raw data description

For the sake of space, we decided not to show data for restaurants here. Very similar things as in 1 can be observed.

A menu item looks like this:

Listing 3: Menu item information in JSON.

```
{'CategoryId': 17717,
 'ToppingsSet': None,
 'TraditionalName': None,
 'Price': 27.0,
 'DeliveryTokenWeight': 1.0,
 'Allergens': {'IsVegetarian': False,
  'IsVegan': False,
  'IsNutsFree': False,
  'IsCeliac': False},
 'Notes': '',
 'MenuId': 8340,
 'IsRecommended': False,
```

```
 'IsEnabled': True,
 'IsEnabledForLocation': True,
 'IsActive': True,
 'OrderNo': 2,
 'MenuItemLocations': [],
 'MenuItemId': 56452,
 'MenuItemName': 'Jelly Teddy Bears',
 'Description': 'Made with strawberry and pink peppercorn',
 'ImageUrl': 'cakesandbubbles/menus/Jelly Teddy Bears-56452.jpg',
 'RestaurantId': 'f6a98cf3-ff2f-4422-a514-7b5248039763',
 'RestaurantCode': 'cakesandbubbles',
 'ChangeTime': '0001-01-01T00:00:00-00:01',
 'ChangedByUser': None,
 'DefaultMenu': True}
```

2.4. Foodhub

(This source was mainly processed by Amer Alkojjeh).
Relevant code, within folder `Task2_Getting_Data`:

- `Foodhub_restaurant_menu.ipynb` is the script used for collecting the data.
- `/Data/foodhub_with_menu.json` is the processed data file for this source.

2.4.1. Collection process

Again, the the procedure used here was very similar as for the other sources. However, data for a different region of London was carried out, namely for Stockwell, with post code "SW9 9TN". The structure here was also similar as above, as there was one response necessary to get a list of restaurants, and further, per-restaurant requests for more detailed info, such as menu data.

2.4.2. Raw data description

Again, for the sake of space we refrain from including raw data on the restaurants here, however as the script saves no auxiliary data we decided to show raw data of a menu item (with some elements removed):

Listing 4: Menu item information in JSON.

```
{'id': 10685776,
 'host': 'afandinalebanesecusine.co.uk',
 'item_addon_cat': '7395670',
 'name': 'Falafel Special',
 'description': 'Deep fried mixture of ground chickpeas and broad beans
     with spices served with hummus, salad, lebanese bread and tahina
    sauce.',
 'information': None,
 'price': '10.50',
```

```
8
9   ...
10
11  'coupon_allowed': 1,
12  'collection_discount_allowed': 1,
13  'online_discount_allowed': 1,
14  'item_code': None,
15  'added': '2018-08-10 15:39:24',
16  'user_id': 'yohanpradeep',
17  'page': 'add_item_insert',
18  'modified': '2019-12-04 22:05:53',
19  'modified_by': 'Fabinbaptist',
20  'modified_page': 'update_description',
21  'vat': '1',
22  'exclude_free': 1,
23  'second_language_name': '',
24  'second_language_description': '',
25  'printer': 0,
26  'section': 0,
27  'is_print_label': '0',
28  'is_vat_included': '1',
29  'food_type': 'NONE',
30  'sections': None}
```

3. Task 3: Creating the Ontology

3.1. External Ontologies

We present here three existing ontologies that cover suitable vocabularies for our domain.

1. BBC food ontology:[1]
   This is a lightweight ontology for publishing data about recipes, food, menus, courses and occasions that they might suitable for. Whilst it originates in a specific BBC use case, the Food Ontology should be applicable to a wide range of recipe data publishing across the web. Two of the vocabularies we are interested in this ontology are: food, menu.
2. International Contact Ontology:[2]
   This ontology provides basic classes and more detailed properties for representing international street addresses, phone numbers and emails. These vocabularies build an important part of the vocabularies we need to model in our domain.
3. Good Relation Ontology:[3]
   This is a lightweight ontology for exchanging e-commerce information, namely data about products, offers, points of sale, prices, terms and conditions, on the Web.

[1] https://www.bbc.co.uk/ontologies/fo
[2] http://ontology.eil.utoronto.ca/icontact.html#d4e383
[3] http://www.heppnetz.de/ontologies/goodrelations/v1.html

Some of these terms like payment methods, legal name and description share a similar semantic of concepts we want to present in our ontology.

3.2. Building Process

We tested Protégé[4], both its web and desktop applications, furthermore WebVOWL. After first tests, we decided to use the Protégé desktop version as the main tool to build our ontology, since it supports most of the requirements needed to complete this task. For example, it supports the option to merge different ontologies. It is faster compared to the web version and allows us to apply reasoning, a feature that is missing from the web application. In addition, we used WebVOWL for visualizing interaction throughout and after the creation process. This gave us a fast and better understanding of the whole structure of the reused ontologies and their connections, which on the one hand eased the pruning step, and on the other hand led us to a few mistakes that were still hidden after creating our ontology, and enabled us to correct them.

We considered the general process of building an ontology provided in the task description, and we followed the next steps in a iterative and non-linear process to complete this task:

- We started to prune each of the three reused ontologies separately and kept those vocabularies needed for our domain.
- We merged the pruned parts into one ontology and set the URI of our ontology as `http://metafood.org` before creating the new elements.
- We added the new classes and the object and data properties to complete the ontology. Furthermore, we set the domain and the range of each property.
- We added some constraints (type and cardinality constraints). Example: An address consists of street, street number, postal code at most.
- We applied the reasoner from time to time, to validate the correctness of our ontology and derive additional knowledge.
- For each new element, we added the required comments and labels.
- We exported the ontology in the Turtle format.

Listing 5: Excerpt of MetaFood Ontology exported as Turtle.

```
1 #################################################################
2 #    Classes
3 #################################################################
4
5 ###  http://metafood.org#Breakfast
6 food:Breakfast rdf:type owl:Class ;
```

[4] `https://protege.stanford.edu`

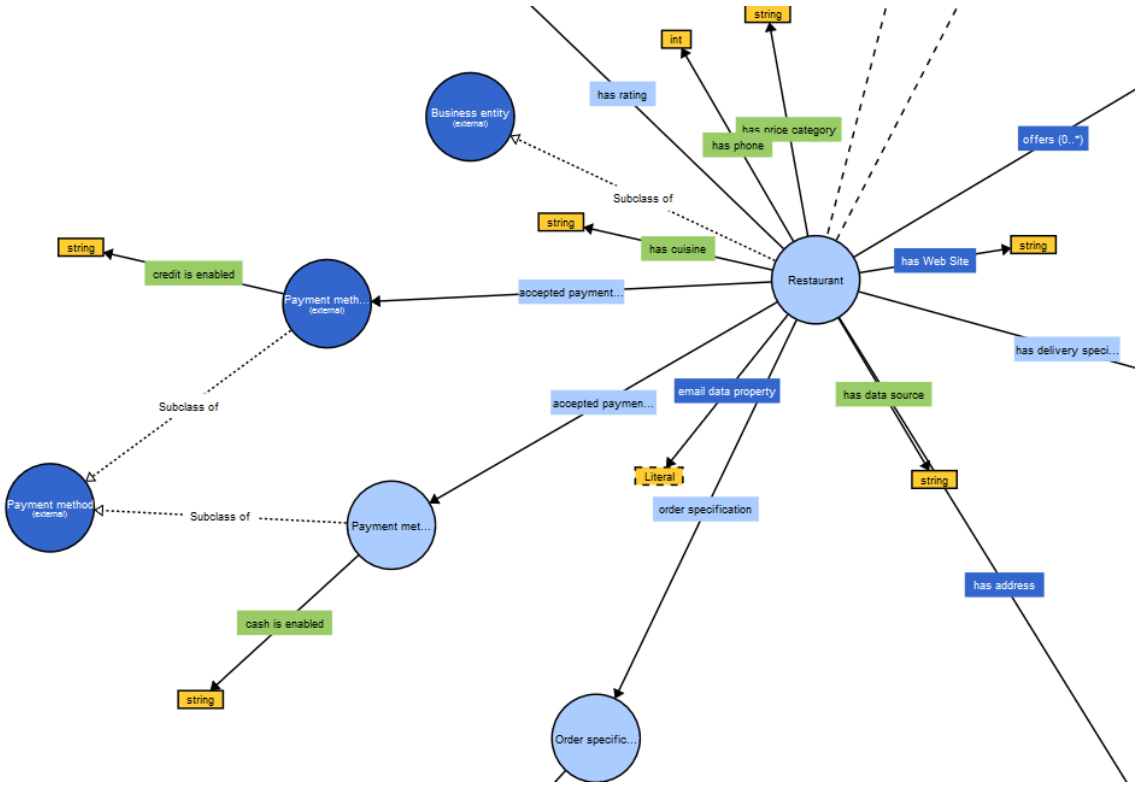

Figure 2: Snapshot of MetaFood Ontology

```
7                rdfs:subClassOf <http://purl.org/ontology/fo/Food> ;
8                rdfs:comment "The subclass of fo:Food represents the
                    first meal of the day that is by a restaurant." ;
9                rdfs:label "Breakfast" .
10
11
12 ###  http://metafood.org#DeliverySpecification
13 food:DeliverySpecification rdf:type owl:Class ;
14                           rdfs:subClassOf owl:Thing ;
15                           rdfs:comment """This class represents
                                characteristics relates to a delivery
                                service offered by a restaurant.
16 Examples: delivery charge, delivery time""" ;
17                           rdfs:label "Delivery specification" .
18
19
20 ###  http://metafood.org#MainDish
21 food:MainDish rdf:type owl:Class ;
22               rdfs:subClassOf <http://purl.org/ontology/fo/Food> ;
23               rdfs:comment "The subclass of fo:Food represents a meal
                    eaten around midday which is offered by a restaurant
                    ." ;
24               rdfs:label "Main dish" .
```

4. Task 4: Creating a Knowledge Graph

After we obtained the first version of our ontology, we started with the task of creating our Knowledge Graph. Since we have tabular data, we used Open Refine to generate the RDF representation from our datasets following these steps:

- we imported the dataset (csv file), created a project, and started building the skeleton as follows:
  - We set the Base URI of our Ontology: `http://metafood.org`
  - We imported all parts included in our ontology and set up their prefixes.

Prefix	URI
food:	`http://metafood.org#`
fo:	`http://purl.org/ontology/fo/`
gr:	`http://purl.org/goodrelations/v1#`
geo:	`http://www.w3.org/2003/01/geo/wgs84_pos#`
ic:	`http://ontology.eil.utoronto.ca/icontact.owl#`

  - We set up a mapping skeleton between the columns' names of our dataset and the vocabularies from the imported ontologies.
  - We used columns with unique values to build up URIs for the resources.
    For example, we used the unique id values of restaurants to build up URIs for resources such as restaurant, address, delivery specification, order specification etc. Moreover, we used the row number values to build up URIs for the menu resource.
    In this step, we used GREL to generate the required URIs. Unfortunately, it was not possible to preview a generated URI value after setting the GREL expression. That is why we needed to check it by previewing the RDF output.
  - We previewed the RDF representation from time to time to see how the mapping is taking shape and adapted the skeleton accordingly.
  - We exported the RDF representation as Turtle.
  - To replicate the process for the other datasets, we exported the operation history of building our skeleton as JSON file.

- While following the above steps:
  - We needed to revise our ontology from time to time and adapt it in order to complete this task.

Workload division: Since our datasets share the same schema, we worked together to build our RDF skeleton and test it. Then, each group member applied the defined mapping on his own dataset to generate the corresponding Knowledge Graph.

Listing 6: Excerpt of the RDF skeleton in JSON.

```
"baseUri": "http://metafood.org/",
      "rootNodes": [
        {
          "nodeType": "cell-as-resource",
          "columnName": "url",
          "expression": "'restaurant/' + value",
          "isRowNumberCell": false,
          "links": [
            {
              "uri": "http://purl.org/goodrelations/v1#legalName",
              "curie": "gr:legalName",
              "target": {
                "nodeType": "cell-as-literal",
                "columnName": "name_x",
                "expression": "value",
                "isRowNumberCell": false
              }
            },
            {
              "uri": "http://purl.org/goodrelations/v1#description",
              "curie": "gr:description",
              "target": {
                "nodeType": "cell-as-literal",
                "columnName": "description_x",
                "expression": "value",
                "isRowNumberCell": false
              }
            },
```

Listing 7: Excerpt of the Knowledge Graph in Turtle generated from FoodHub dataset.

```
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix fo: <http://purl.org/ontology/fo/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ic: <http://ontology.eil.utoronto.ca/icontact.owl#> .
@prefix gr: <http://purl.org/goodrelations/v1#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix food: <http://metafood.org#> .


<http://metafood.org/restaurant/afandinalebanesecusine.co.uk> a food:
   Restaurant;
  food:acceptedPaymentCashMethod <http://metafood.org/PaymentMethodCash
     /afandinalebanesecusine.co.uk>;
  food:acceptedPaymentCreditMethod <http://metafood.org/
     PaymentMethodCreditCard/afandinalebanesecusine.co.uk>;
  food:hasCuisine "['Burgers', 'Kebab']";
  food:hasDeliverySpecification <http://metafood.org/
```

```
    DeliverySpecification/afandinalebanesecusine.co.uk>;
17  food:hasOrderSpecification <http://metafood.org/OrderSpecification/
    afandinalebanesecusine.co.uk>;
18  food:hasDeliverySpecification <http://metafood.org/
    DeliverySpecification/afandinalebanesecusine.co.uk>;
19  food:hasPhone "02077377577"^^<http://www.w3.org/2001/XMLSchema#int>;
20  food:hasRating <http://metafood.org/rating/afandinalebanesecusine.co.
    uk>;
21  ic:hasAddress <http://metafood.org/address/afandinalebanesecusine.co.
    uk>;
22  food:hasRating <http://metafood.org/rating/afandinalebanesecusine.co.
    uk>;
23  ic:hasWebSite "afandinalebanesecusine.co.uk";
24  gr:description "Afandina Lebanese Cusine in London offers a variety
    of Great Tasting Grilled Chicken,Kebabs,Rice Dishes,Seafood
    Dishes & Many More Delights";
25  gr:legalName "Afandina Lebanese Cusine";
26  food:hasOrderSpecification <http://metafood.org/OrderSpecification/
    afandinalebanesecusine.co.uk>;
27  gr:offers <http://metafood.org/breakfast/afandinalebanesecusine.co.uk
    >, <http://metafood.org/mainDish/afandinalebanesecusine.co.uk>,
28    <http://metafood.org/menu/0>, <http://metafood.org/menu/1>, <http
        ://metafood.org/menu/2>,
29    <http://metafood.org/snack/afandinalebanesecusine.co.uk> .
```

5. Task 5: Knowledge Graph to Triple Store

We used Graph DB to load our Knowledge Graph into Triple Store and created seven construct queries using SPARQL in order to enrich our Knowledge Graph before creating the required queries in task 6.

In order to do that, first we had to create a new repository, set its ID (metafood) and the Base URL to `http://metafood.org#`. Then, we imported the Knowledge Graph as created in the previous task and set the Base IRI to `http://metafood.org#`.

The construct queries are:

- We assigned hasTotalDeliveryPrice property for each menu. Its value is the sum of menu price (Price of a menu offered by a restaurant) and delivery charge (the amount of money it takes for a delivery service to deliver the order).
- We assigned hasTotalDeliveryTime property for each restaurant. Its value is the sum of waiting time (how long an order takes after being picked up to be ready for delivery) and delivery time (typical delivery time to an address, that it takes for a delivery service to deliver the order).
- We assigned hasAreaCode property for each restaurant. Its value is extracted from its phone number.
- We assign hasStars property with value "***" for each restaurant which has a rating value (It takes values between 0 and 5 to evaluate a restaurant on the

delivery service website) more than 4.5 and a rating count (how many users has rated a restaurant on the delivery service website) of more than 2.

- We assign hasStars property with value "**" for each restaurant which has a rating value between 4.5-3 and a rating count of more than 2.
- We assign hasStars property with value "*" for each restaurant which has a rating value less than 3 and rating count more than 2.
- We assigned acceptedPaymentCashCreditMethod property for each restaurant, that accepts both cash and credit payment methods.

Listing 8: Construct query in SPARQL to assign hasTotalDeliveryPrice property for each menu.

```
PREFIX food: <http://metafood.org#>
PREFIX fo: <http://purl.org/ontology/fo/>
PREFIX gr: <http://purl.org/goodrelations/v1#>

construct {?menu food:hasTotalDeliveryPrice ?totalDeliveryPrice .}
   where {
select distinct ?menu ((?deliveryCharge +?price) AS ?totalDeliveryPrice
   )  where {
    ?restaurant food:hasDeliverySpecification ?ds .
        ?ds food:deliveryCharge ?deliveryCharge .

    ?restaurant gr:offers ?menu.
    ?menu a fo:Menu ;
          food:hasPrice ?price .
}

}
```

Listing 9: Construct query in SPARQL to assign hasStars property with value "***" for the corresponding restaurants.

```
PREFIX food: <http://metafood.org#>
PREFIX gr: <http://purl.org/goodrelations/v1#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT  {?rating food:hasStars ?stars .} where {
        ?restaurant food:hasRating ?rating .
    ?rating food:hasRatingCount ?RatingCount .
    ?rating food:hasRatingValue ?value ;
       FILTER (?value >=4.5 && ?RatingCount >= 2)
        BIND (xsd:string("***") as ?stars)
}
```

6. Task 6: SPARQL Queries

Before starting with this task, we inserted the RDF graphs which resulted from the queries that we had written in the previous task using the INSERT function (SPARQL).

We translated the competency questions into SPARQL queries as follows: There are names in competency questions such as postal code and city that represent entities with exact the same terms or similar ones in our knowledge graph. These names in the competency questions could be combined with values, such as postal code of X. Such values represent values of the corresponding entities. We took the structure of the skeleton that has been defined into consideration when we wanted to traverse from one entity to another in order to construct the required connections between these entities in a SPARQL query. For the values of such entities, we used Filter functions in a SPARQL query to retrieve them. To translate terms like highest, lowest that are combined with a name or an adjective such as rated or price, we used ORDER BY combined with LIMIT in SPARQL.

We created nine queries (Q) in SPARQL to answer different competency questions (CQ) that have been written and edited through the last steps. The Queries were created based on the foodHub Knowledge Graph.

- CQ 1: Which delivery service delivers food X in less than 30 minutes?
  Q 1: It returns the delivery service name with restaurants names that deliver pasta in less than 30 minutes.

- CQ 2: Which delivery service delivers free drinks with Food X?
  Q 2: It returns the delivery service name with restaurants names which deliver menus that include chicken or crispy with free drink.

- CQ 3: At which delivery service can I order food from the X highest rated restaurant in district Y?
  Q 3: It returns the delivery service name with the five highest rated Indian or Chinese restaurants in London that have a rating value of more than 2.

- CQ 4: At which delivery name service can I order food X with price less than Y pounds?
  Q 4: It returns the delivery service name with restaurants that deliver a Pilav Rice menu with a total price of less than 6 pounds included the delivery price.

- CQ 5: At which delivery service which delivers to postal code X can I pay by credit card/cash?
  Q 5: It returns the delivery service name that accepts credit card as payment method, delivers to postal code ”SW9 9” from five highest rated restaurants with a rating value more than 2.

- CQ 6: Which delivery service delivers the highest rated pizza Y to postal code X?
  Q 6: It returns the delivery service name that delivers pizza ”Margherita” to ”SW9 9” postal code with the total delivery time from the three highest rated pizzerias.

- CQ 7: Which delivery service delivers gluten-free/lactose-free/halal/kosher food to postal code X?
  Q 7: It returns the delivery service name that delivers Halal food to "SW9 9" postal code offered by the highest rated restaurants.

- CQ 8: Which delivery service delivers food X and drink Y with the cheapest price in district Z?
  Q 8: It returns the delivery service name that delivers burger and alcohol drinks from the ten cheapest restaurants in London.

- CQ 9: Which delivery service delivers high rated food X to district Y and accepts cash as payment method?
  Q 9: It returns the delivery service name that delivers spicy food and accepts cash as a payment method from the five highest rated restaurants in the Brixton area.

Workload division: Each team member developed three of the previous queries. Amer Alkojjeh developed queries 1, 2 and 3. Thassilo Gadermaier developed queries 4, 5 and 6. Philip Klaus developed queries 7, 8 and 9.

Listing 10: Which delivery service delivers free drinks with Food X?

```
PREFIX food: <http://metafood.org#>
PREFIX ic: <http://ontology.eil.utoronto.ca/icontact.owl#>
PREFIX gr: <http://purl.org/goodrelations/v1#>
PREFIX fo: <http://purl.org/ontology/fo/>

select ?deliveryService ?restaurantName ?phone ?menuName ?description ?price where {

 ?restaurant gr:legalName ?restaurantName.
 ?restaurant food:hasPhone ?phone.
 ?restaurant food:hasDataSource ?deliveryService.

 ?restaurant gr:offers ?menu.
 ?menu a fo:Menu ;
       food:hasPrice ?price ;
       gr:legalName ?menuName ;
       FILTER (REGEX(?menuName, ".*Chicken*.") || REGEX(?menuName, ".*Crispy*."))

     ?menu gr:description ?description ;
       FILTER (CONTAINS(?description, "drink")  || CONTAINS(?description, "Drink") )

}
```

Listing 11: Which delivery service delivers food X and drink Y with the cheapest price in district Z?

```
PREFIX food: <http://metafood.org#>
PREFIX ic: <http://ontology.eil.utoronto.ca/icontact.owl#>
PREFIX gr: <http://purl.org/goodrelations/v1#>
PREFIX fo: <http://purl.org/ontology/fo/>

select ?deliveryService ?restaurantName ?phone ?RatingValue ?cuisine ?
    menuName ?price where {
     ?restaurant food:hasPhone ?phone.
     ?restaurant gr:legalName ?restaurantName.
     ?restaurant food:hasDataSource ?deliveryService.

     ?restaurant ic:hasAddress ?address .
     ?address ic:hasCity ?city;
     FILTER (REGEX(?city, "London"))

         ?restaurant food:hasCuisine ?cuisine ;
        FILTER (CONTAINS(?cuisine, "Burger")  || CONTAINS(?cuisine, "
           Alcohol"))

     ?restaurant gr:offers ?menu.
     ?menu a fo:Menu ;
           gr:legalName ?menuName ;
           food:hasPrice ?price .

     ?restaurant food:hasRating ?rating .
     ?rating food:hasRatingValue ?RatingValue .

} ORDER BY ?price limit 10
```

7. Task 7: Creating the Application

For the task 7 we decided to implement a simple webapp for visualizing some of our SPARQL-queries.

7.1. User interface

The main idea behind our user interface (see figure 3) was that a user should be able to search for products easily and should be able to refine his search request intuitively by applying several filters. Therefore we divided the UI into 3 main sections:

1. **Product and filter selection**
2. **Selected products**
3. **Product suggestions**

At page-loading all available products are queried. The user can view these products by expanding the product section under the ”Select Products” link. By clicking one or more products, the products are added to the ”Selected Products” card and a HTTP-Request is sent to our server which contains all selected products together wit

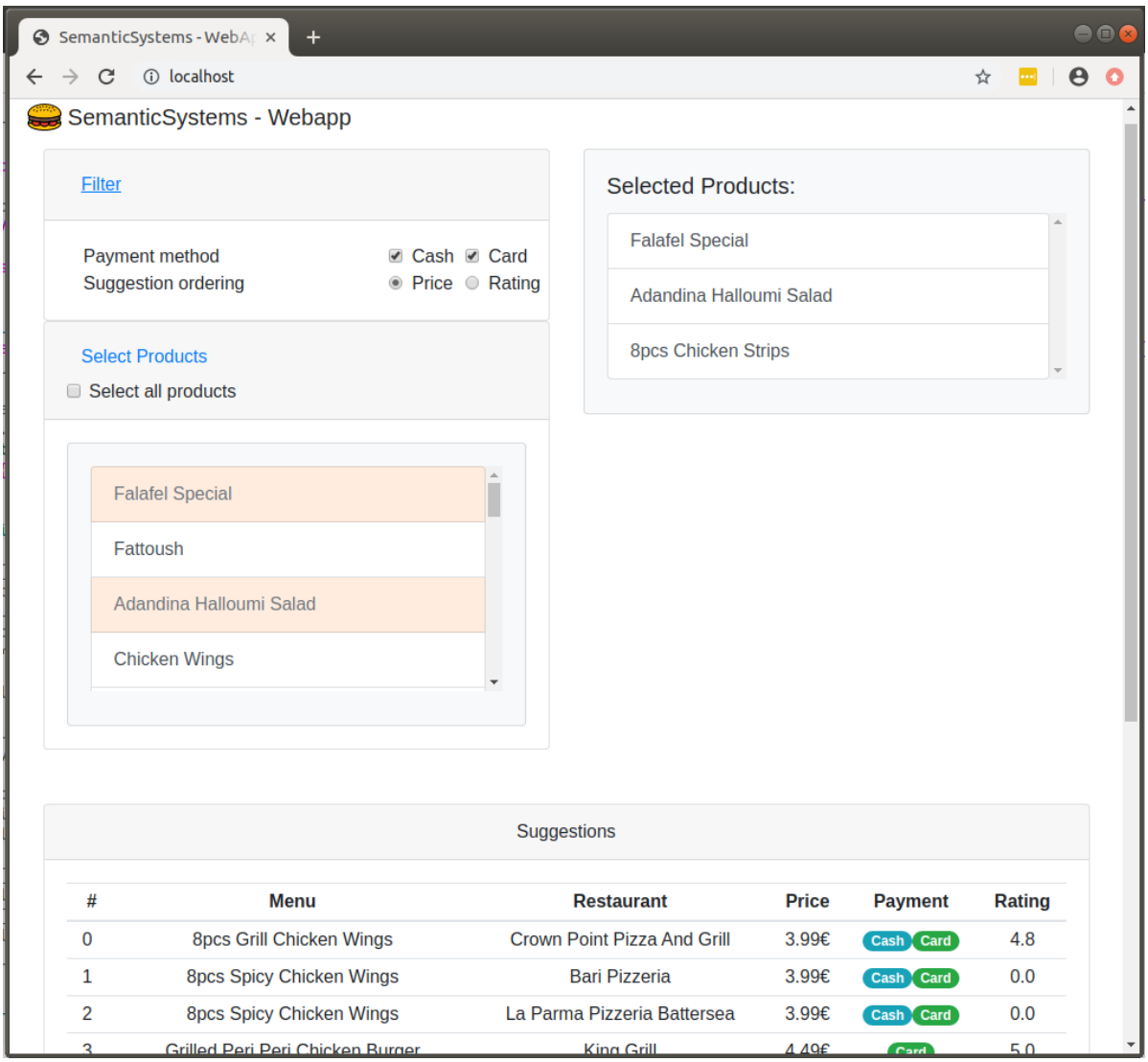

Figure 3: Screenshot of out SemanticSystems - Webapp

all active filters. The server then splits each product string into single words, sends a SPARQL-HTTP-request for retrieving information about all products which contain at least one of the single words and which matches the actual filters to the RDF4J database-server. After receiving a HTTP-Response from the database-server, it sends the result as a HTTP-Response back to the Browser which displays the result in the ”Suggestions” section. Each time the user selects or deselects a product, or switches between the filters, a new request is made. In a real application the filters would just be applied to already retrieved products in the UI. But for demonstration purposes of several SPARQL-Request-Features, the filters are used to directly alter, build and execute new SPAQRL queries. To refine the search the following filters can be applied:

- **Payment method**

Retrieve products from restaurants which offers at least ”Cash”, ”Card”, or both payment methods

- **Suggestion ordering**
  Retrieve the products from the database ordered by either ”Price” or ”Rating”

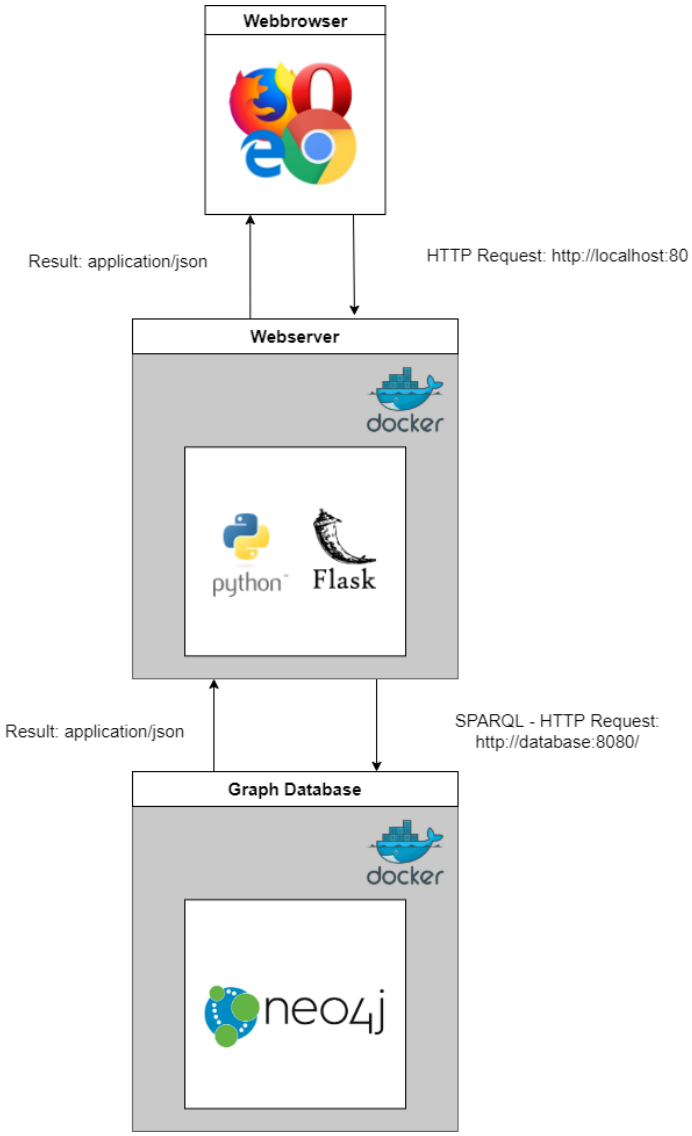

Figure 4: The technology-scheme describing our webapp

7.2. Technological details

7.2.1. Workflow

Basically our webapp follows the workflow described in 4. The user navigates to the webapp in the browser served by the web-server and selects products or applies filters.

Then a HTTP-GET request for retrieving product suggestions is send to the web-server. The server constructs the appropriate SPARQL-Query string on the fly, using a simple self-implemented SPARQL-Query-Builder function. Then it sends a HTTP-GET request with the URL-encoded SPARQL-Query to the Graph Database which answers the request with an HTTP-Response containing the product suggestions JSON-formatted in the response-body. At the end the server responses the JSON data back to the web-app which displays the results to the user.

7.2.2. The webserver

The webserver runs in a Docker container and is implemented using Python 3 and the Flask web application framework `https://www.palletsprojects.com/p/flask/`. At the first incoming HTTP-Request, the server sends our RDF data stored in `/sem_system/Task4_KG/foodHub_KG.ttl` to the Graph Database which receives and stores the RDF data internally.

7.2.3. The Graph Database

For the application we used the neo4j database `https://neo4j.com/` for storing our RDF data and requesting results using SPARQL-HTTP-Queries. In our presentation of the project, we showed our web-app communicating with the GraphDB database. However, as we wanted to dockerize our application for the final submission, and the GraphDB-Docker-Image did not suite our requirements, we switched to using neo4j.

7.2.4. Running the application

Attention: To run our application Docker and Docker-Compose have to be installed!

As all of our components are dockerized it is just pretty simple to build and run our web application. To run the app just navigate to the sub folder `/sem_system/webapp/` and execute the following commands:

1. docker-compose build
2. docker-compose up

Afterwards the web-app should be hosted on `http://localhost:80`

A. Additional Data

Here is the original documentation of the data acquired by the original 4th Member of our project, Lukas Anzinger. We leave it as is, for documentation purposes.

A.1. Deliveroo

A.1.1. Collection process

Deliveroo has thousands of restaurants under contract but it is only possible to order from nearby restaurants. Since for the project it was enough to have a data set that consists of about 100 restaurants, I decided to only get the restaurants and menus that deliver to a certain region. We decided to use Stockwell, a district in London, which has the postcode *SW9 9TN*.

After entering the postcode on the Deliveroo website, a list of all restaurants that deliver to Stockwell is shown—grouped by certain categories like *Fastest delivery* or *Featured* (see `https://deliveroo.co.uk/restaurants/london/stockwell?postcode=SW9+9TN`). Clicking on one of the restaurants shows the menu of the restaurant, i.e. the meals, their descriptions and prices.

The goal was now to get a list of all restaurants and their menus for the given postcode, i.e. to scrape the website. I decided to use Python and the Requests library (`http://python-requests.org`). I also thought about using the Scrapy framework (`https://scrapy.org`) but dismissed the idea because of the added complexity over a simple solution based on the Requests library.

The workflow for obtaining the data is as follows:

1. Request the site that lists all restaurants for a given region/postcode.
2. Extract URLs to the restaurants from the response HTML code
3. Request the restaurant URLs
4. Parse the HTML code and extract data to answer the *competency questions*.

A.1.2. Raw data description

The Deliveroo website is unfortunately *not* written as a single-page application[5] with a JSON API for communicating with the backend. Instead, the HTML code is rendered on the server and returned to the user. In such a case, the HTML code returned from Extracting data requires to parse the HTML code in such a case. I use the parsel library (`https://pypi.org/project/parsel/`) for this purpose which allows to use CSS selectors for efficient and concise data extraction from the HTML code. In Listing 12 the HTML response that contains the links to the restaurants' menus is shown. To extract the links from the HTML code, the CSS selector `ul[class^='HomeFeedGrid'] a::attr('href')` is used.

Listing 12: Excerpt of the HTML code that lists all restaurants for a region.

```
1 <ul class="HomeFeedGrid-...">
2     <li class="...">
3         <div class="...">
4             <a href="/menu/london/brixton/souvlaki-street-conor-mills?
                  day=today&amp;postcode=SW99TN&amp;time=ASAP">
```

[5] See also `https://en.wikipedia.org/wiki/Single-page_application`

```
            (...)
        </div>
    </li>
    (...)
</ul>
```

The extracted restaurant links are then requested one after another. The restaurant page contains the menu information in *machine-readable form* (JSON) which makes extracting the menu information quite convenient.

Listing 13: Restaurant information in JSON, embedded in the HTML code.

```
<script type="application/json">
{
   (...)
   "restaurant" : {
      "open" : true,
      "uname" : "477-pret-a-manger-brixton",
      "post_code" : "SW98HE",
      "phone_numbers" : {
         "primary" : "+442077372799",
         "secondary" : null
      },
      "description" : "Sandwiches, salads, baguettes, wraps and more.
         Freshly prepared in our shop kitchen everyday.",
      "accepts_allergy_notes" : false,
      "newly_added" : false,
      "opens_at" : "08:00",
      "custom_allergy_note" : null,
      "customer_collection_supported" : false,
      "delivery_fee" : null,
      "enabled_fulfillment_methods" : [
         "DELIVERY"
      ],
      "price_category" : null,
      "id" : 62899,
      "street_address" : "417 Brixton Road",
      "closes_at" : "18:00",
      "neighborhood" : "Brixton",
      "city" : "London",
      "name" : "Pret A Manger ",
      "brand_name" : "Pret",
      "name_with_branch" : "Pret A Manger  - Brixton",
      (...)
   },
   (...)
   "menu" : {
      (...)
      "items" : [
         {
            "price" : " 1 .40",
            (...)
            "description" : "A freshly baked all-butter croissant."
```

```
41          },
42          {
43              "price" : " 1 .70",
44              (...)
45              "description" : "Flaky pastry filled with 5 berry Jam and
                    folded."
46          },
47          (...)
48      }
49      (...)
50  }
```

The JSON is parsed and interesting fields are extracted for further processing.