Final Report of Semantic Web about

Food

Aspir Ahmet
Tobias Dick
Daniel Egger
Stefan Kaufmann

14.02.2019

Contents

1 Introduction

This knowledge-graph recipe application serves as an online recipe-look-up service. Either by entering certain ingredients, selecting health labels and diet labels or combing both of the previously mentioned, the request will yield according results. A web-overlay ensures the interaction with regular users.

1.1 Use Case

This application can be used by everyone who is looking for new and refreshing recipes. However it is especially useful for people who are doing a diet or for people whose nutrition is restricted by certain allergies.

People who just want to try out new recipes with certain ingredients can easily add those ingredients to the search. After the search process they will receive only recipes containing all the given ingredients as a result.

1.2 Data Sources

On the search for useful data sources for a lot of different recipes we came across an API called Edamam. Edamam organizes the worlds food knowledge in one database and is free to use. The database contains over 1.7 million different recipes which can easily be crawled by a simple java program. This way it provides us with tons of different recipes and a lot of information for each recipe. This information contains obvious properties like the recipe name, cook time and ingredients, but it also contains exact health and diet labels for every recipe. Since the given information is so rich, Edamam has been our main data source since the beginning of the project.

Anther data source we have been using for a large part of the semester is a small online data set. This data set is free to use and contains about 520 different recipes. However the information provided for each recipe is very minor. This finally led to the exclusion of the data set from our data sources. The main reason for the exclusion is, that the recipes from the small data set would violate the validation through our SHACL shapes. This however will be explained in more detail in a later section.

In both cases the data was provided in JSON format, which made the parsing and further storing of the recipes pretty easy.

1.3 Ontology

Our first idea was to use schema.org because of its popularity. We were able to find a schema.org class that fitted our intentions perfectly. The main class is `<https://schema.org/Recipe>` which contains its own recipe properties as well as properties of `<https://schema.org/HowTo>`, `<https://schema.org/`

`CreativeWork`> and <`https://schema.org/Thing`>. Given these properties we were able to map most of the recipe attributes, which we use in our application. However not all of them could be mapped to a fitting property. Therefore an extension of the ontology was needed.

The first extension made was targeted at the poperty recipeIngredient. In the recipe class from schema.org the recipeIngredient property only consists of a simple text. This however is not sufficient for our purpose since our application requires us to save recipe ingredients as seen in Figure 1. Unfortunately we were not able to find an existing ontology which would provide this exact structure. Therefore we extended the base vocabulary from schema.org with our own extension for recipeIngredient.

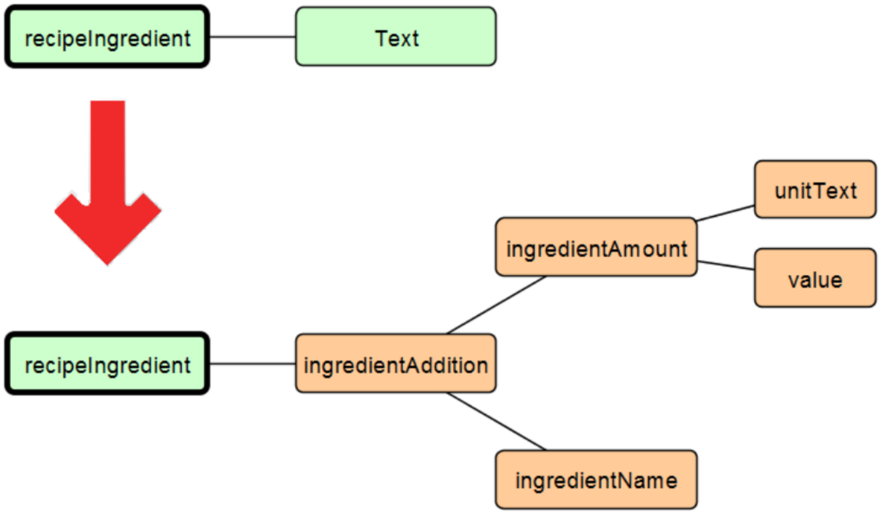


Figure 1: recipeIngredient Extension

The recipe class of schema.org also does not contain a property for a recipes cuisine. For our application or respectively our crawling process this property makes a lot of sense, since we crawl for different recipes dependent on their cuisine. This led to the second extension of our used ontology. In contrast to the first extension we were able to find an existing ontology which contains a cuisine property. This new ontology is an explicit food ontology and can be found at <`https://www.bbc.co.uk/ontologies/fo`>.

Given these extensions our final ontology can be seen in Figure 2.

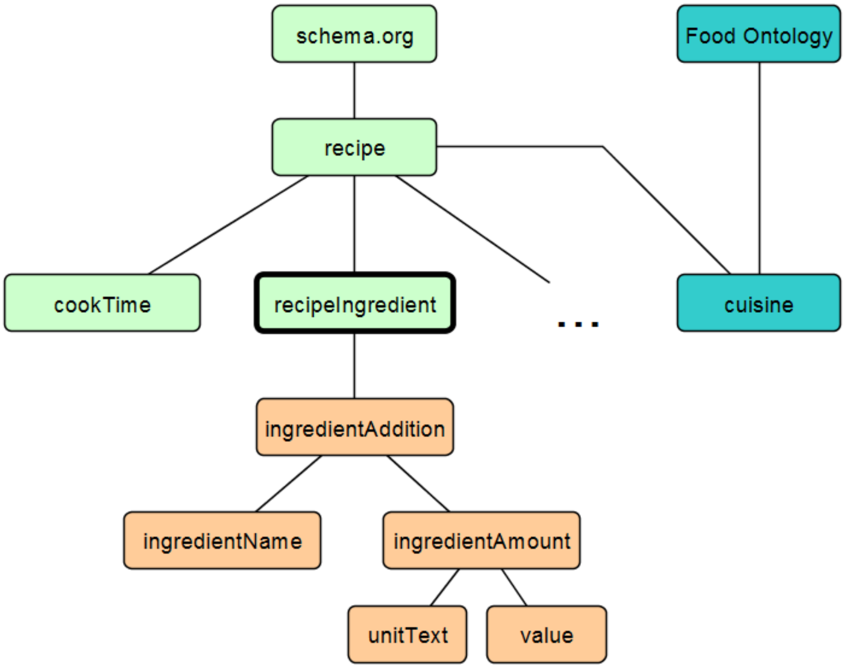
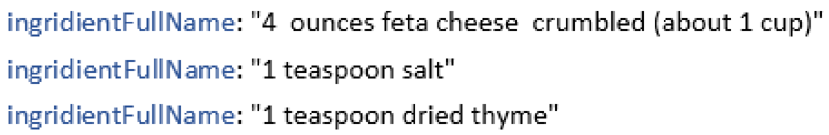


Figure 2: The final Ontology

2 Enrichment

2.1 Information Extraction - NLP

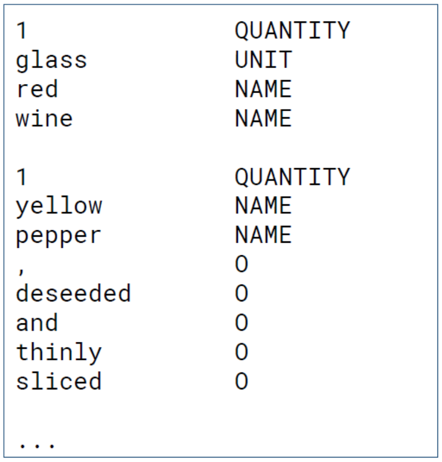
As discussed in the previous section all of our recipes come from the Edamam API. However this API provides the recipe ingredients as simple ingredient-lines, which can be seen in Figure 3. But the extension we have made to our ontology regarding recipeIngredient requires us to save every ingredient with a name, an unit text and an amount. Therefore we have to extract the needed information from this ingredient-lines. This however is not that simple because every line can look different, which makes a common parser useless.



```
ingridientFullName: "4  ounces feta cheese  crumbled (about 1 cup)"
ingridientFullName: "1 teaspoon salt"
ingridientFullName: "1 teaspoon dried thyme"
```

Figure 3: Three Ingredient-Lines

To achieve a valid parsing we used natural language processing (NLP). NLP is a field of machine learning that seeks to understand human languages and provides specific tools to extract pieces of information, such as NER (named-entity recognition). We have used the Stanford NER tagger on a tokenized version of the corresponding ingredient-line. An example for such a tokenized and tagged ingredient-line is shown in Figure 4.



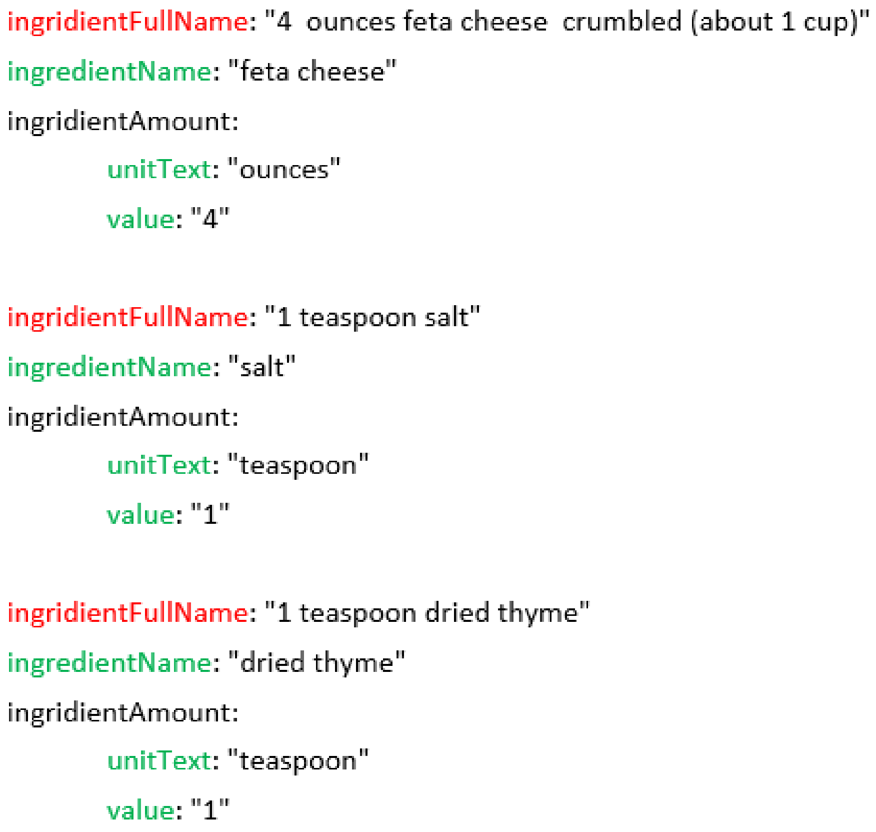
```
1               QUANTITY
glass           UNIT
red             NAME
wine            NAME

1               QUANTITY
yellow          NAME
pepper          NAME
,               O
deseeded        O
and             O
thinly          O
sliced          O

...
```

Figure 4: Tokenized Ingredient-Line

Since we needed to achieve a high accuracy with the named-entity recognition, so that our ingredients would be parsed the right way, we manually created a training model which was specific for our data set. An example of what we achieved with this strategy is shown in Figure 5.



```
ingridientFullName: "4  ounces feta cheese  crumbled (about 1 cup)"
ingredientName: "feta cheese"
ingridientAmount:
        unitText: "ounces"
        value: "4"

ingridientFullName: "1 teaspoon salt"
ingredientName: "salt"
ingridientAmount:
        unitText: "teaspoon"
        value: "1"

ingridientFullName: "1 teaspoon dried thyme"
ingredientName: "dried thyme"
ingridientAmount:
        unitText: "teaspoon"
        value: "1"
```

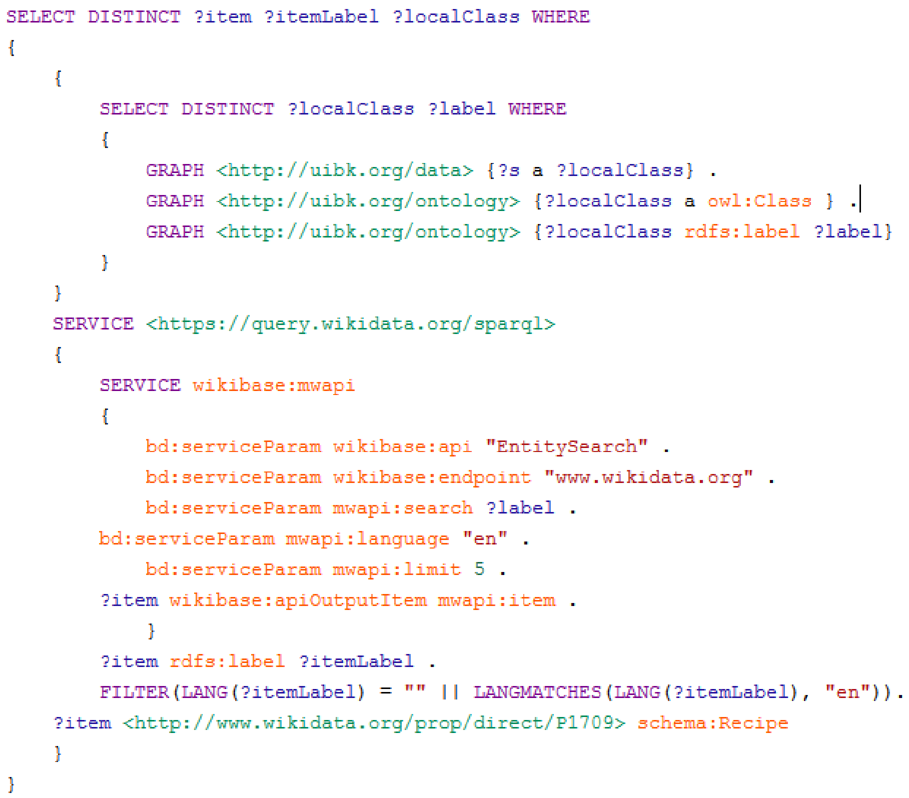
Figure 5: Achived Example Results with NLP

2.2 LOD Linking

2.2.1 Data Alignment and the Federated Query

The goal of data alignment is to align our data types with already existing and widely used types. Prime-example for this is Wikidata. Using Wikidatas SPARQL-Endpoint it is possible to find equivalent classes to locally used classes. In this case only schema:Recipe was aligned but it is totally possible to find an alignment for all locally used classes and integrate the Wikidata classes into the ontology in an automated fashion. This requires certain steps such as: Load all yielded Class-Codes from Wikidata to a lookup-table, using a script integrate all loaded Class-Codes into existing ontology via “Protege”. This was not necessary hence only a demonstrative federated query was included.

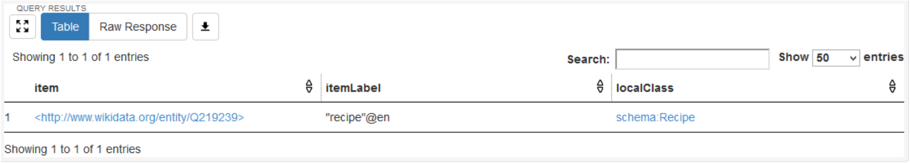
The used federated query is shown in Figure 6:



```
SELECT DISTINCT ?item ?itemLabel ?localClass WHERE
{
    {
        SELECT DISTINCT ?localClass ?label WHERE
        {
            GRAPH <http://uibk.org/data> {?s a ?localClass} .
            GRAPH <http://uibk.org/ontology> {?localClass a owl:Class } .
            GRAPH <http://uibk.org/ontology> {?localClass rdfs:label ?label}
        }
    }
    SERVICE <https://query.wikidata.org/sparql>
    {
        SERVICE wikibase:mwapi
        {
            bd:serviceParam wikibase:api "EntitySearch" .
            bd:serviceParam wikibase:endpoint "www.wikidata.org" .
            bd:serviceParam mwapi:search ?label .
        bd:serviceParam mwapi:language "en" .
            bd:serviceParam mwapi:limit 5 .
        ?item wikibase:apiOutputItem mwapi:item .
            }
        ?item rdfs:label ?itemLabel .
        FILTER(LANG(?itemLabel) = "" || LANGMATCHES(LANG(?itemLabel), "en")).
    ?item <http://www.wikidata.org/prop/direct/P1709> schema:Recipe
    }
}
```

Figure 6: Our Federated Query

The first part of the query focuses on getting all locally used classes and their labels '?localClass' '?label'. Once these are known using the keyword 'SERVICE' an external query-service is consulted (Wikidata in this case). On top of the normal usual SPARQL-Query service Wikidata offers a Search-API (MediaWiki SearchAPI) which comes in handy since the labels of the local classes can simply be entered in that API which then returns the according search-results. The part 'SERVICE wikibase:mwapi' returns the Wikidata objects into '?item' which is then further used. The API-Request can be modified by changing the according 'bd:serviceParam'. When finally receiving entries as '?item' from the API a simple query is submitted to find the labels which are then displayed at the local triple store that requested the SERVICE. The last line '?item <`http://www.wikidata.org/prop/direct/P1709`> schema:Recipe' is to filter to our locally used class schema:Recipe since Wikidata offers a direct equivalence property to 'schema.org' classes. In general the rest of this query could have been avoided, but for reasons mentioned earlier it is useful to leave it in anyway. The result of the query can be seen in Figure 7:



Showing 1 to 1 of 1 entries

Search: Show 50 entries

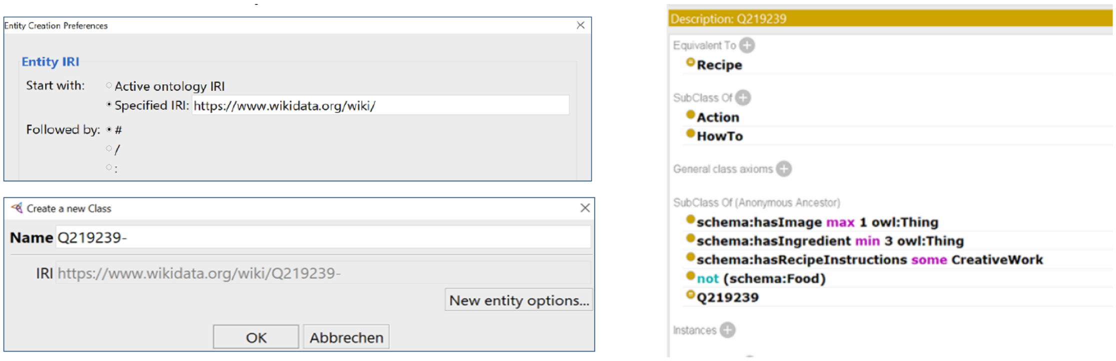
	item	itemLabel	localClass
1	<http://www.wikidata.org/entity/Q219239>	"recipe"@en	schema:Recipe

Showing 1 to 1 of 1 entries

Figure 7: The result of the federated Query

2.2.2 Linking with Entity of Wikidata

As seen in the previous section we get the corresponding identifier "Q219239" from Wikidata as a result from our federated Query. Now we want to link our schema:Recipe with Wikidata (more specific: with the given identifier). To do so we used the program "Protege" (Figure 8) :



Start with: ○ Active ontology IRI
● Specified IRI: https://www.wikidata.org/wiki/
Followed by: ● #
○ /
○ :

Name Q219239-
IRI https://www.wikidata.org/wiki/Q219239-

Equivalent To
Recipe

SubClass Of
Action
HowTo

SubClass Of (Anonymous Ancestor)
schema:hasImage max 1 owl:Thing
schema:hasIngredient min 3 owl:Thing
schema:hasRecipeInstructions some CreativeWork
not (schema:Food)
Q219239

Figure 8: Create a link to Wikidata

First we create a new class in "Protege" while specifying the IRI in the preferences to link it to Wikidata. We name the entity we want to create the same as the Q-identifier from Wikidata (since it is the representation of the Wikidata entity).

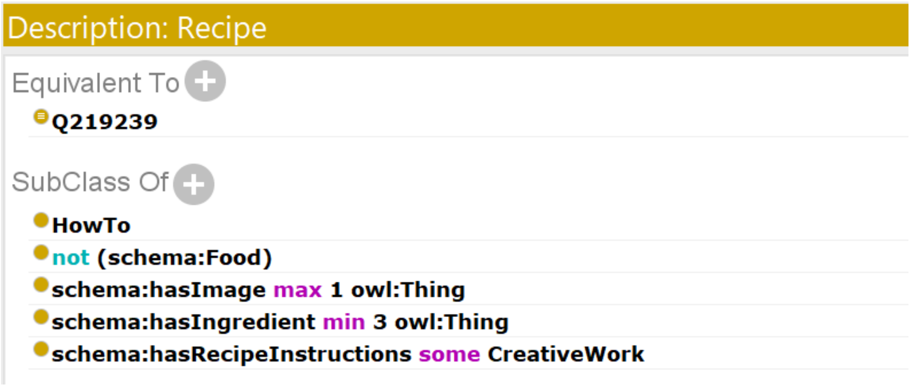
After that we now set the to classes in "Protege" as equal. As shown on the right side in Figure 8 the subclasses are automatically inferred by the program.

So after the linking is done we also started a reasoner to infer some extra constraints for the classes.

Together with the OWL axioms (described in the next section) we exported and used the made changes to the ontology.

2.3 OWL Axioms

We used OWL axioms in our project to associate class and property IDs with complete specifications of their characteristics. We also used it to give other logical information about classes and properties.
Each class axiom in OWL contains a collection of more-general classes and a collection of local property restrictions in the form of restriction constructs. The restriction construct gives the local range of a property, how many values are permitted, and/or a collection of required values. The OWL axioms which we defined for our ontology can be seen in Figure 9 and are directly saved in the ontology.



Description: Recipe

Equivalent To

Q219239

SubClass Of

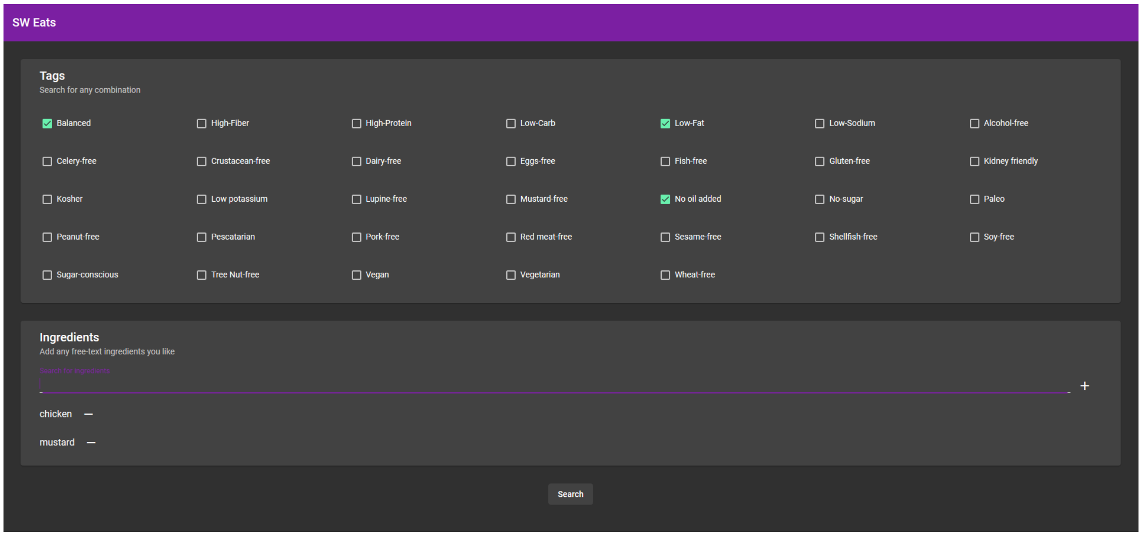
HowTo
not (schema:Food)
schema:hasImage max 1 owl:Thing
schema:hasIngredient min 3 owl:Thing
schema:hasRecipeInstructions some CreativeWork

Figure 9: OWL Axioms

3 Application

3.1 General Description

In the following picture (Figure 10) the main page for the user can be seen:



SW Eats

Tags

Search for any combination

☑ Balanced	□ High-Fiber	□ High-Protein	□ Low-Carb	☑ Low-Fat	□ Low-Sodium	□ Alcohol-free
□ Celery-free	□ Crustacean-free	□ Dairy-free	□ Eggs-free	□ Fish-free	□ Gluten-free	□ Kidney friendly
□ Kosher	□ Low potassium	□ Lupine-free	□ Mustard-free	☑ No oil added	□ No-sugar	□ Paleo
□ Peanut-free	□ Pescatarian	□ Pork-free	□ Red meat-free	□ Sesame-free	□ Shellfish-free	□ Soy-free
□ Sugar-conscious	□ Tree Nut-free	□ Vegan	□ Vegetarian	□ Wheat-free		

Ingredients

Add any free-text ingredients you like

Search for ingredients

chicken —

mustard —

Figure 10: Main Page for our application

Here the user can choose between different tags for the search which will be considered by our application. As already mentioned, we are using Edamam as our main data source and thus our tags shown in the picture follow directly the available tags (diet and health labels) stated by Edamam. The tags are set and unset by simple checkboxes.
In the other half of our starting page the input field for the ingredients can be found. Here the user can state single ingredients and add them to list of ingredients which is displayed directly below the input field. The user is also able to delete previously stated ingredients via the "–" button.

Currently our database stores approximately 4500 different recipes which are considered when performing a search through our application.

3.2 Architecture

Basically our architecture consists of three main components shown in Figure 11:

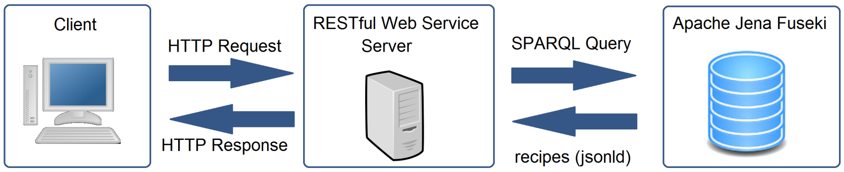


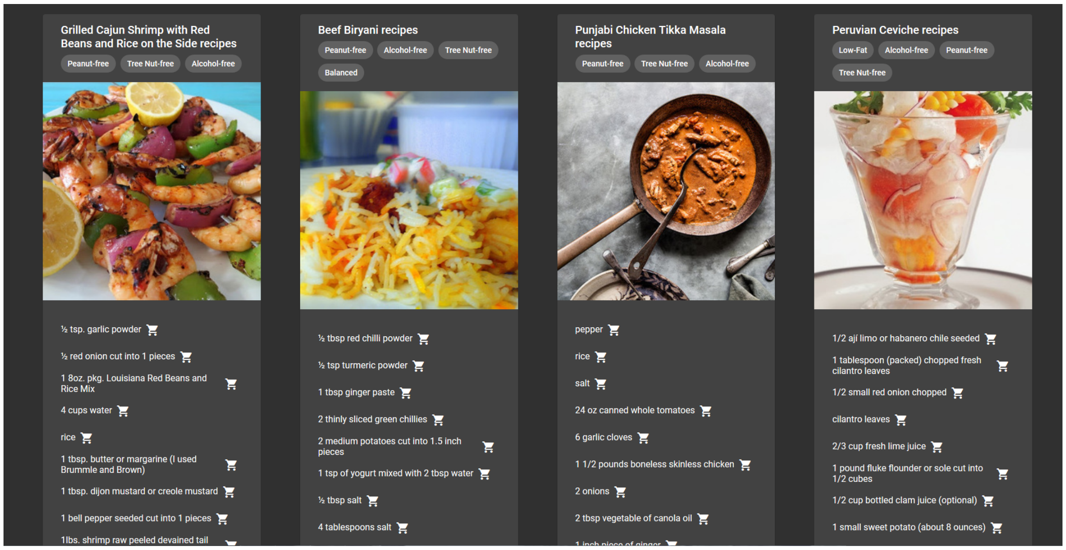
Figure 11: Basic architecture of our application

As seen in the picture the client (where the main page is displayed) sends a HTTP request with the desired tags and a list of the chosen ingredients to our RESTful Web Service Server. The RESTful Server parses the HTTP request (extracts the ingredients and tags from the HTTP request) and forms a dynamic SPARQL query, which is then sent to the Apache Jena Fuseki Server, where our triples are stored. The Fuseki Server then executes the query and responds with the desired data in Turtle-format (with the keyword DESCRIBE). We then take this result and parse it into jsonld which we hand back to the RESTful Web Service Server. This server forms a HTTP response with the results from the query and sends it back to the client where the results will be displayed properly.

3.2.1 Client

In this section we want to talk a little bit about the front-end of our application. The front-end is designed with “Angular” and is available on port 4200. Angular is a platform that makes it easy to build applications with the web. Angular combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges.

We performed an example request without any tags selected. The result is displayed in Figure 12 to get an impression what our front-end looks like:



Grilled Cajun Shrimp with Red Beans and Rice on the Side recipes

Peanut-free Tree Nut-free Alcohol-free

- ½ tsp. garlic powder
- ½ red onion cut into 1 pieces
- 1 8oz. pkg. Louisiana Red Beans and Rice Mix
- 4 cups water
- rice
- 1 tbsp. butter or margarine (I used Brummle and Brown)
- 1 tbsp. dijon mustard or creole mustard
- 1 bell pepper seeded cut into 1 pieces
- 1lbs. shrimp raw peeled devained tail

Beef Biryani recipes

Peanut-free Alcohol-free Tree Nut-free Balanced

- ½ tbsp red chilli powder
- ½ tsp turmeric powder
- 1 tbsp ginger paste
- 2 thinly sliced green chillies
- 2 medium potatoes cut into 1.5 inch pieces
- 1 tsp of yogurt mixed with 2 tbsp water
- ½ tbsp salt
- 4 tablespoons salt

Punjabi Chicken Tikka Masala recipes

Peanut-free Tree Nut-free Alcohol-free

- pepper
- rice
- salt
- 24 oz canned whole tomatoes
- 6 garlic cloves
- 1 1/2 pounds boneless skinless chicken
- 2 onions
- 2 tbsp vegetable of canola oil
- 1 inch piece of ginger

Peruvian Ceviche recipes

Low-Fat Alcohol-free Peanut-free Tree Nut-free

- 1/2 ají limo or habanero chile seeded
- 1 tablespoon (packed) chopped fresh cilantro leaves
- 1/2 small red onion chopped
- cilantro leaves
- 2/3 cup fresh lime juice
- 1 pound fluke flounder or sole cut into 1/2 cubes
- 1/2 cup bottled clam juice (optional)
- 1 small sweet potato (about 8 ounces)

Figure 12: An example search

The user gets a nice looking result with the title of the recipe, the stored tags for each recipe, a picture which gives the user a first impression what the meal might look like after preparation and a list of ingredients.
Here we can also observe that directly after each listed ingredient there is a symbol to click on. When the user clicks on this symbol he is redirected to an online store which is called "freshdirect" where he or she can buy the corresponding ingredient. This is handled via a potentialAction (more specific: SearchAction).

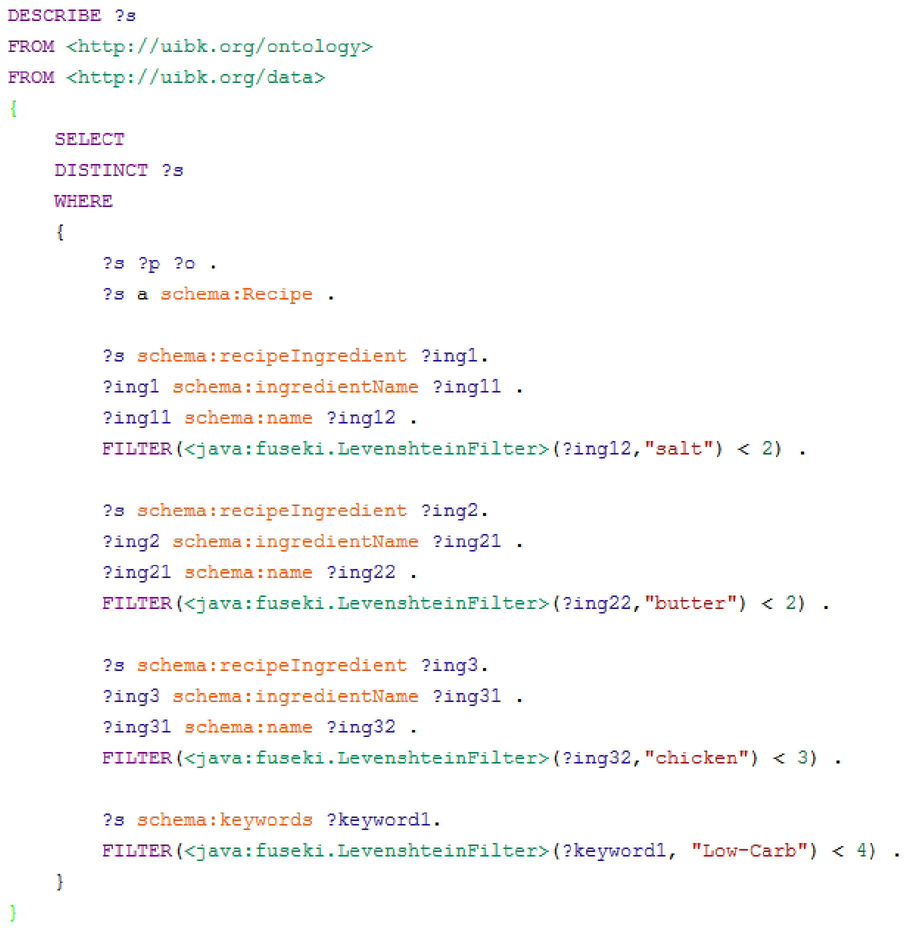
3.2.2 RESTful Web Service Server and Apache Jena Fuseki

The backend part of our application starts with the RESTful Web Service Server which is available on port 8080. This server handles the HTTP request generated by the Client. It simply parses the desired ingredients and also the selected tags out of the HTTP request and hands it over to the second part of the backend, which is responsible for communicating with the Apache Jena Fuseki triple store.

The second part of the backend gets the parsed input and forms a dynamic SPARQL query out of it. The generated Query is sent to the Apache Jena Fuseki Server which executes the query (Apache Jena Fuseki is located at port 3030). The result is then obtained and we parse it into a jsonld result which is then passed back to the front-end, where the results are displayed.

Levenshtein Filter We also implemented a Levenshtein Filter for the Fuseki Server so we can catch similar ingredients (e.g. it doesn't matter if an ingredient is written in lowercase or uppercase). This is implemented via the Levenshtein distance. This distance is a string metric for measuring the difference between two sequences or expressed more simple: the minimum single-character edits between to strings. So for the upper/lower case example the distance would be 1.

the dynamic SPARQL query The goal of this query is, to find entries in the triple store that match the requirements. These requirements are based on selected ingredients and tags. From these user-inputs a dynamic query is generated to get the desired schema:Recipe entries. The general idea is to find the matching subjects to a simple 'DESCRIBE' in the query is used to reconstruct the original recipe in its entirety. Since 'DESCRIBE' in Fusekis SPARQL yields a TTL-format a conversion to JSON-LD is essential since the results can be passed to any web-front-end without much modification of the data. An example application query (with ingredients: salt, butter, chicken and tag: Low-Carb) is shown in Figure 13:



```
DESCRIBE ?s
FROM <http://uibk.org/ontology>
FROM <http://uibk.org/data>
{
    SELECT
    DISTINCT ?s
    WHERE
    {
        ?s ?p ?o .
        ?s a schema:Recipe .

        ?s schema:recipeIngredient ?ing1.
        ?ing1 schema:ingredientName ?ing11 .
        ?ing11 schema:name ?ing12 .
        FILTER(<java:fuseki.LevenshteinFilter>(?ing12,"salt") < 2) .

        ?s schema:recipeIngredient ?ing2.
        ?ing2 schema:ingredientName ?ing21 .
        ?ing21 schema:name ?ing22 .
        FILTER(<java:fuseki.LevenshteinFilter>(?ing22,"butter") < 2) .

        ?s schema:recipeIngredient ?ing3.
        ?ing3 schema:ingredientName ?ing31 .
        ?ing31 schema:name ?ing32 .
        FILTER(<java:fuseki.LevenshteinFilter>(?ing32,"chicken") < 3) .

        ?s schema:keywords ?keyword1.
        FILTER(<java:fuseki.LevenshteinFilter>(?keyword1, "Low-Carb") < 4) .
    }
}
```

Figure 13: An example application query

The first line is ’DESCRIBE ?s’: This returns a TTL-Format RDF Graph reconstruction of a specified resource. In particular of the subject ?s which should ultimately be a schema:Recipe. The ’FROM’ specifies the data and ontology graph. Now the ingredients need to be somehow looked up in the graph. First all possible values of ’?s’ must be a ’schema:Recipe’. Ingredients are linked to the recipe by the predicate ’schema:recipeIngredient’. This class has 3 classes in itself: ’schema:ingredientName’, ’schema:ingredientAmount’, ’schema:potentialAction’. Further the name of the ingredient can be found in the class ’schema:ingredientName’. Once the ’schema:ingredientName’ object has been found a simple ’schema:name’ will contain the actual name of an ingredient. The line ’FILTER(¡java:fuseki.LevenshteinFilter>(?ing12,”salt”)

< 2) .' applies a filter to match the ingredient name to a given search-word which is 'butter' in this case. Because user-error is sometimes inevitable the search-word is not directly matched to the entries. Instead a custom-made function 'LevenshteinFilter' extends the current SPARQL of the Fuseki to allow an error-margin when passing the search-words to the query. In this particular case any ingredient names will match satisfying the condition of 'LevenshteinFilter(ingredientNameIntripleStore,searchWord) < 2'. The number '2' is automatically generated for the initial query and custom for every search-word and it simply depends on the length of the search-word. The same method is applied for other ingredients and tags.

3.3 Data Crawling

In this section we want to talk about our pipeline we implemented for data crawling. The pipeline itself is shown in Figure 14:

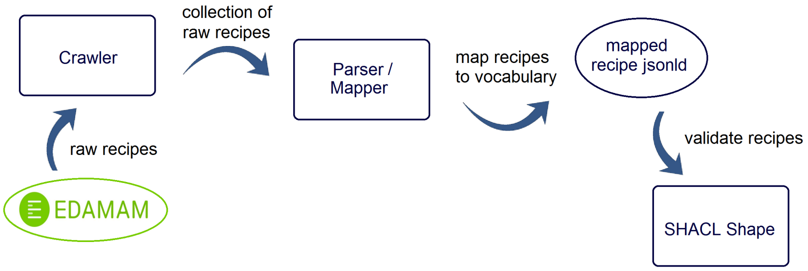


Figure 14: The data crawling pipeline

The pipeline startes with the main data source Edamam and our crawler. We basically state different cuisines as described before and the crawler goes through these categories and sends requests for each cuisine to the Edamam API.

After we obtained the collection of the raw recipes we hand it over to the parser/mapper stage of our pipeline. In this stage we map our raw data to our vocabulary and form it into jsonld so we have a mapped-recipe-jsonld at the end of this stage.

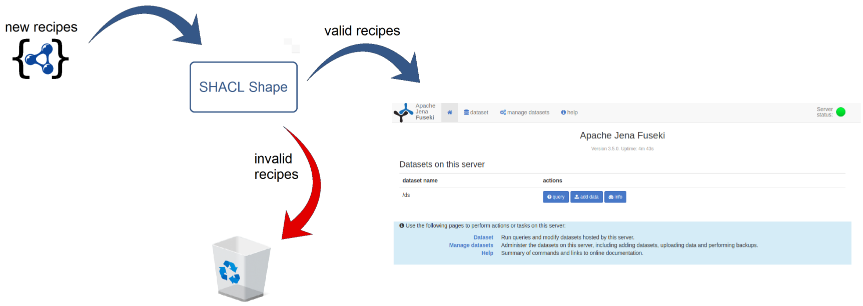
As the last stage of the pipeline we automatically validate our data with the SHACL shapes which are explained in the following section.

3.4 SHACL Shapes

As already mentioned in the first section we use SHACL shapes to validate our data. We do this to guarantee that only valid recipes are uploaded and stored into the Apache Jena Fuseki. Our SHACL shapes check whether the recipes have the correct format and furthermore if the individual properties have the correct data type.

For this validation we have written a java program, which uses the shape and checks it against every recipe file. Any of the recipe files that cause a violation will be made invalid. After being made invalid our program deletes the incorrect recipe from the recipe file and makes the affected recipe file valid again.

If a recipe file does not cause any violation it can safely be uploaded and stored into the Fuseki. The uploading process again is handled by a java program which automatically uploads all correct recipe files. A graphic representation of this process is shown in Figure 15.



Apache Jena Fuseki

Version 3.5.0. Uptime: 4m 43s

Datasets on this server

dataset name	actions
/ds	query add data info

Use the following pages to perform actions or tasks on this server:

Dataset	Run queries and modify datasets hosted by this server.
Manage datasets	Administer the datasets on this server, including adding datasets, uploading data and performing backups.
Help	Summary of commands and links to online documentation.

Figure 15: SHACL Representation

4 Conclusion

The aspects of Semantic Web learned throughout the semester were all used in the project. Some of them are used in the application, while others are an important part of the data crawling process or the preparation of the data. All in all, the project was an interesting introduction to Semantic Web and taught us how to combine and leverage the different aspects of Semantic Web.