Cocktailicious

Isabelle Kuiper 2596565, Femke Volmer 2593159, Romy Vos 2595704

MILESTONE 1 - Application Design

The goal of the application.
The goal of the application is to present cocktail recipes based on the preferences of the user. The user can make a choice in the kind of alcohol. The idea is that the user of the application decides what kind of alcohol the user wants and that the application returns cocktail recipes that contain the users choice of ingredients. Eventually, we want to create a user-friendly interface, where people can easily select their preferences of alcohol and that they get a list of cocktails, that matches their preferences, back. The cocktails will also have a recipe and an ingredients list so that is easy for the user to make the cocktail. To make it even more convenient, we also want to provide a short list with liquor stores where the users can buy the (alcoholic) ingredients that they need.

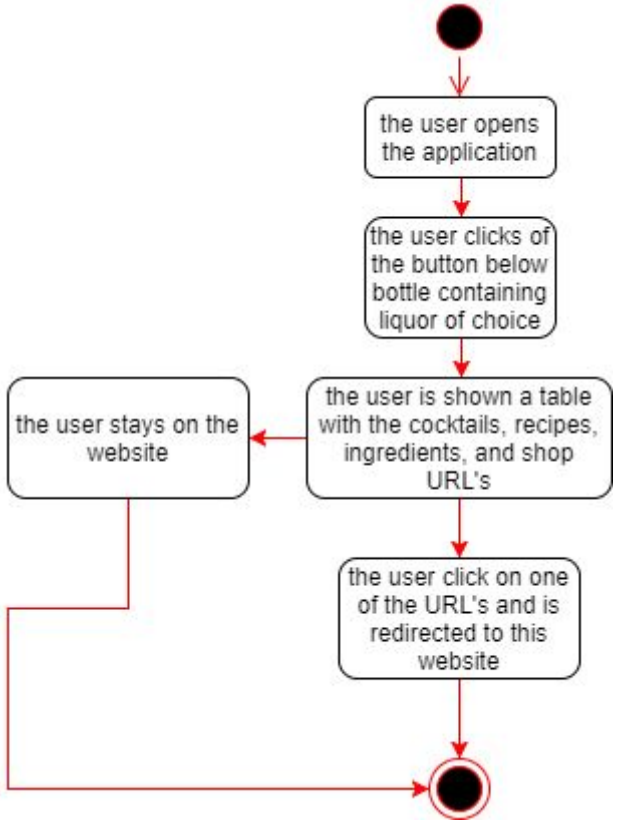
The user: for whom is the application intended?
This application is meant for people who need inspiration for deciding which cocktail they want to drink. We intent to make our application user friendly and easy to use. Because our application involves liquor, we strongly advise the user to only use our application if they are of a legal drinking age. For the people who want to actually make the cocktails themselves, our application also offers the recipes and ingredients. Our application is also useful for someone who has a bottle of a certain liquor leftover and does not know what to do with it, or what kind of cocktails are possible to make with that liquor.

The design: what does it look like and how does it perform?
We want to make our application user friendly and colorful. This means that our web application has to have a good overview. When entering the application, it has to be clear what our application is meant to be used for. We intend to do this with a good title and an introduction of about one sentence. Below that, we want a bar with different bottles of alcoholic drinks. The user can select one of those drinks, and based on that selection, the web application shows the cocktails as a table, containing the cocktails, the recipes, the ingredients, and some website to buy the cocktail ingredients.

Moodboard

Walkthrough: how does the user interact with the application?
When the user visits our application, he/she will read a short explanation of the purpose of the application and how it works. Below the explanation, there is an image of a bar. On the bar, there are placed some bottles, all of a different liquor type. The user selects their preferred liquor, (for example, gin) and when clicking on it, they will see a table with all the cocktails made with their chosen liquor, with for each cocktail a recipe, the needed ingredients and a list of websites to buy the ingredients. Then the user can click further to one of those websites, or they can stay on the page and just make the cocktail or read the recipes.

Activity diagram



MILESTONE 2 - Domain Modeling

Domain and scope of the ontology.

Scope

Our goal is to create a web application that makes it possible to click on an ingredient and then it should return cocktails made with that ingredient. By creating the idea of this application, we had some competency questions in our mind: “I have a little bit of vodka left, what possible cocktails could I make with this?”, or “I always make a mojito. What more cocktails can I make with gin?”. Our application will also give recipes and ingredients for each cocktail, and provide examples of website where the user can buy the ingredients. To make this possible, we have to create a database which contains all cocktails, their recipes and their ingredients, and some liquor stores and their website URL’s.

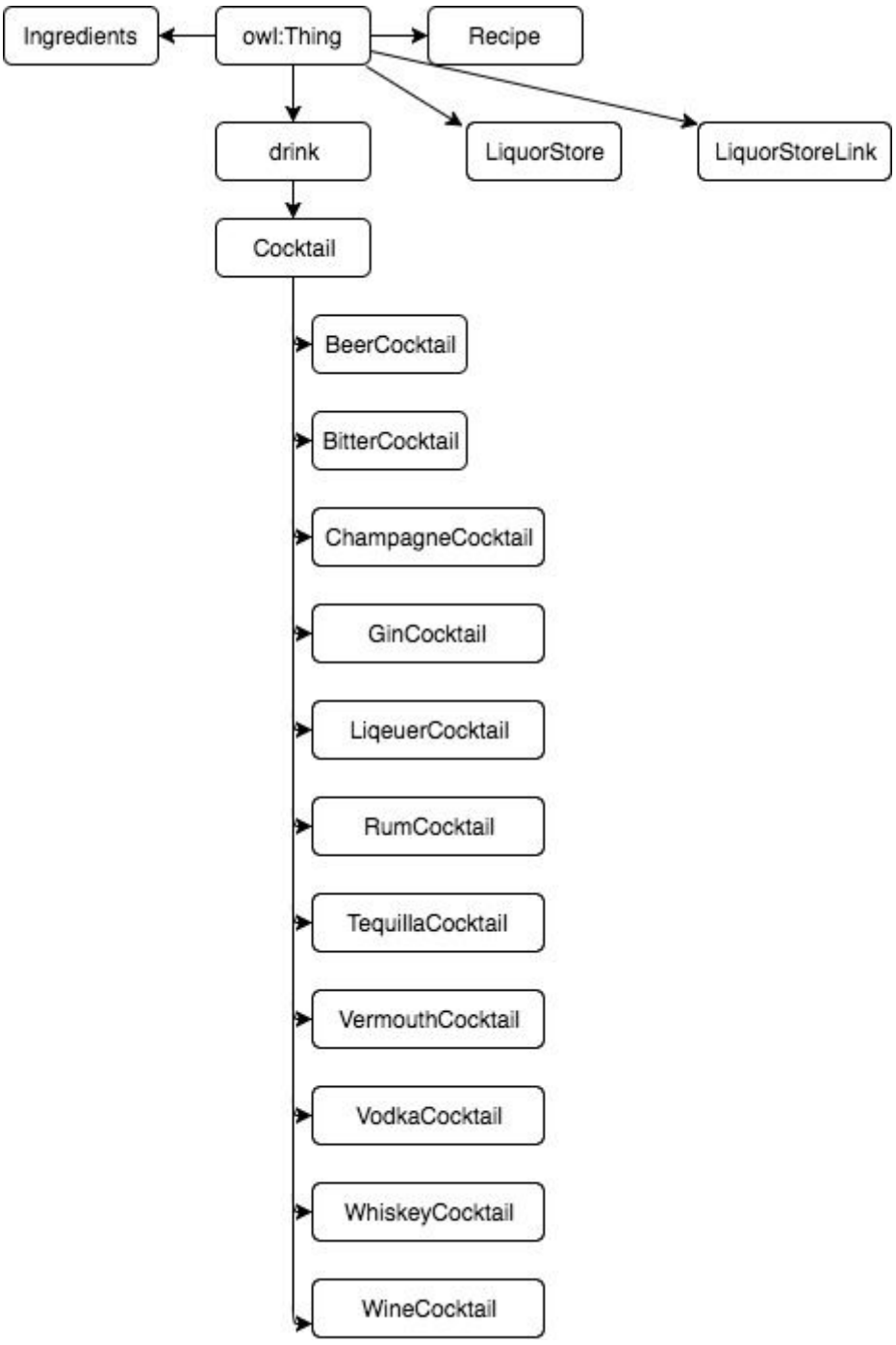
Domain

The domain of our ontology are all the cocktails that exist in our database. These cocktails will get matched to the users by using the user’s preferences. The application will provide the user with the types of cocktails and their recipes and some stores where the user can buy the ingredients.

Methodology

For our ontology we used the top-down methodology. First, we thought of the general idea: we came up with the idea to build our ontology around cocktails. After that, we started to look at the details: we searched for data sets that matched our concept idea. We wanted an application which would help the user with choosing a cocktail, based on preferences involving the ingredients. We also want to provide liquor stores where you can buy the type of alcohol you prefer in your cocktail. We found two external datasets and created an ontology ourselves. From this point, all we had left to do is to integrate the data and build the actual web application.

Conceptualization



The Class Drink has a subClass Cocktail, which has eleven subclasses which are all different types of cocktails.
We have a class Ingredients, this is linked to the class Cocktail with the following equivalent class: Ingredients areIngredientsFor some Cocktail. The properties areIngredientsFor and hasIngredients are each others inverse.
We also have a class Recipe, which is linked to the Class Cocktail with an equivalent class:

Recipe isRecipeFor some Cocktail. Again, the property isRecipeFor and hasRecipe are each others inverse.
The Class LiquorStore is linked to the Class LiquorStoreLink with the equivalent class: LiquorStore hasLink some LiquorStoreLink. The Class LiquorStoreLink uses the inverse property of hasLink, isLinkTo, in the equivalentClass: LiquorStoreLink isLinkTo some LiquorStore.

The above classes are the classes that we created in our own ontology. There are no instances in our own ontology, in our integrated ontology (the final turtle file), all the instances are inferred from the external datasets (DBPedia and WikiData).
The data that we extracted from DBPedia contains cocktail classes (e.g. dbc:Cocktails_with_gin and dbc:Cocktails_with_vodka). We also extracted for all cocktail classes an individual recipe and an individual ingredients (with the DBPedia properties dbr:ingredients and dbr:prep).
From WikiData, we extracted the liquor stores and the website link for all those liquor stores. The exact queries we used for extracting the data can be found in Milestone3.

Ontology with restrictions

Our ontology reuses data from 2 sparql endpoints: http://www.dbpedia.org/sparql and https://query.wikidata.org/. To integrate the data, we had to add class restrictions to the classes in our own ontology, described as above, to match them to the language of the endpoints.
For all the subclasses of our Cocktail class, we reused DBPedia vocabulary by adding the class restriction:
Cocktail dbc:subject value (for example) dbc:Cocktails_with_gin, where the value matches the class we are restricting on.
For our ingredients class, we add the class restriction Ingredients AreIngredientsFor some Cocktail. For our recipe class, we add the restriction Recipe IsRecipeFor some cl:Cocktail. We do want individuals extracted from DBPedia is the Recipe and Ingredients class as well, and since those properties are in the language of our own dataset, we have to create equivalent properties between our own dataset and the DBPedia dataset (see milestone 3).
For our LiquorStore class, we added the restriction LiquorStore hasLink some LiquorStoreLink. For our LiquorStoreLink class, we added the restriction LiquorStoreLink isLinkTo some LiquorStore. This is again the language from our own dataset, so we have to use equivalent properties (and classes) for this as well.

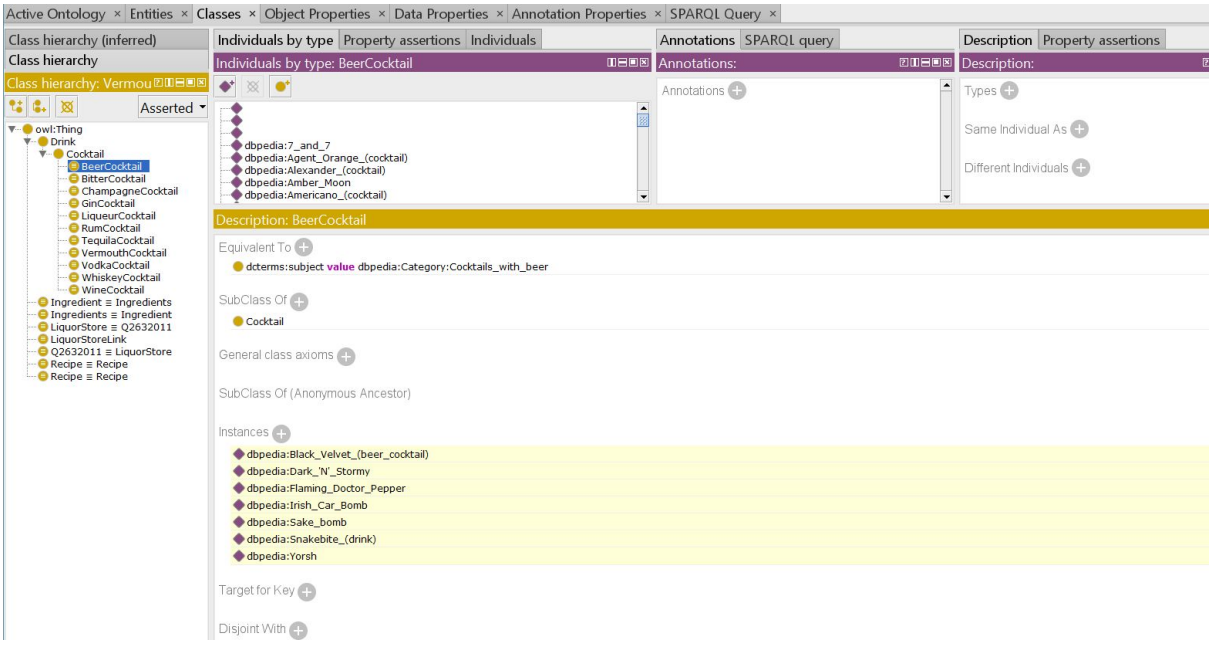
Inferences

We have not used any example data, as can be seen in the turtle file of our ontology. The instances of the classes are all inferred from the external datasets. In our integrated ontology (the final turtle file), we inferred all the cocktails from the DBPedia dataset in the Cocktail class of our own ontology. Then, we inferred all recipes from the DBPedia dataset inside the

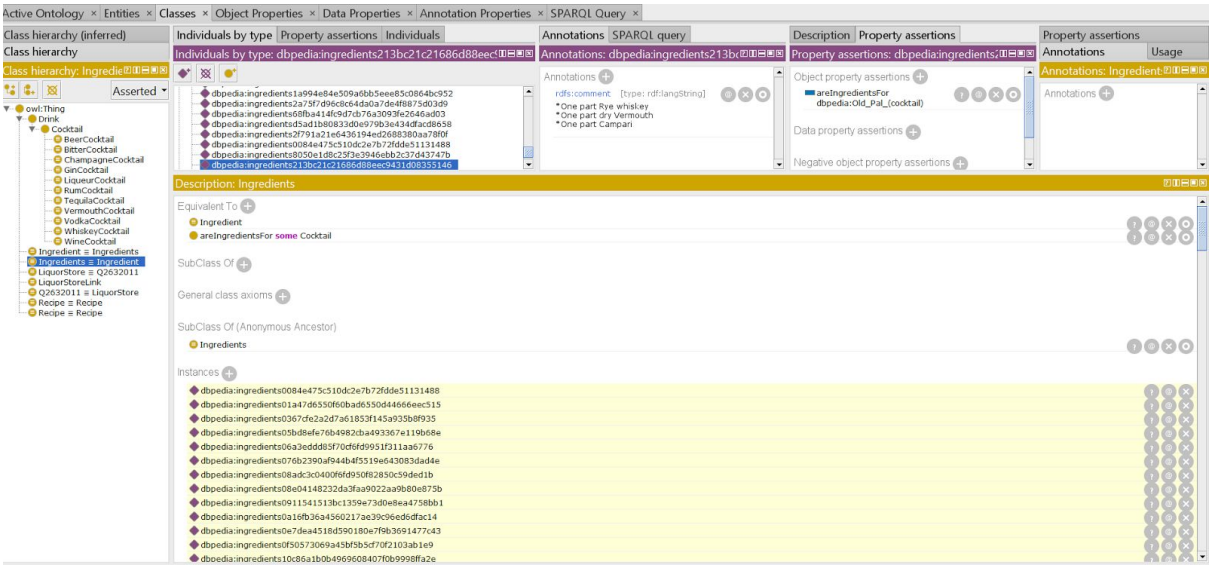
Recipe class of our Ontology and all the extracted ingredients inside the Ingredients class of our ontology: all the cocktails have a dbp:prep followed by a object, and those object are inferred inside our Recipe class. The cocktails also have a dbp:ingredients property, which are inferred inside the Ingredients class.
For the data we extracted from WikiData, the liquor stores are inferred inside the LiquorStore class. Those inferences are not really useful (since it is in WikiData language, so just a bunch of letters and number useless for the user of our web application), which means that it is essential to also infer the LiquorStoreLink, as those are actually useful. Those links are inferred in the LiquorStoreLink class. Details about creating the inferences can be found in the next chapter (Milestone 3).

Screenshots:

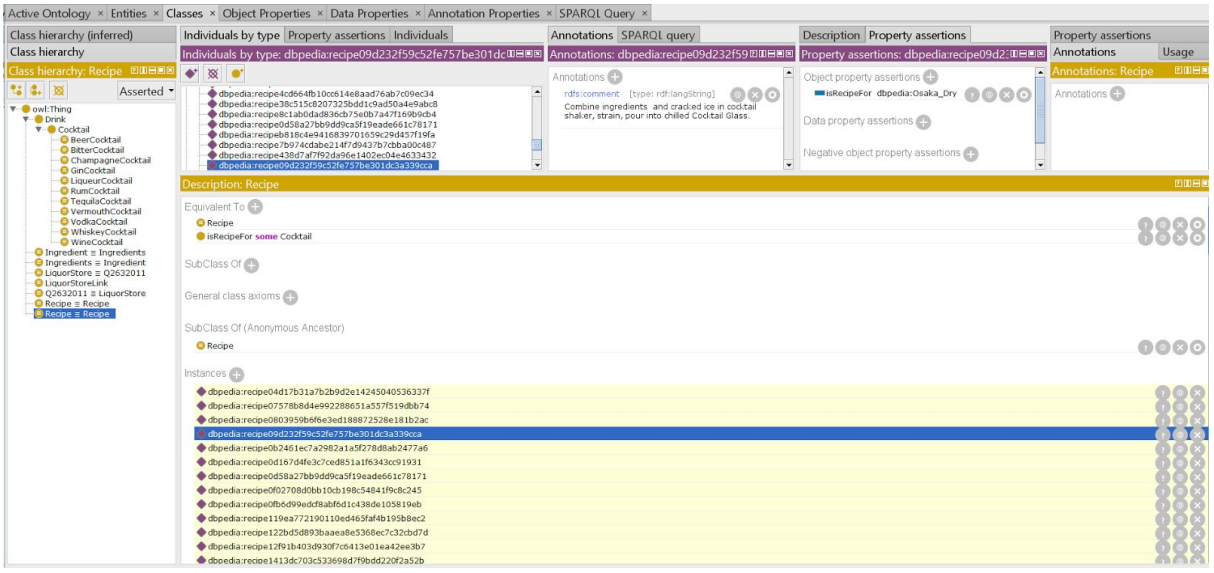
Some examples of the inferences made:



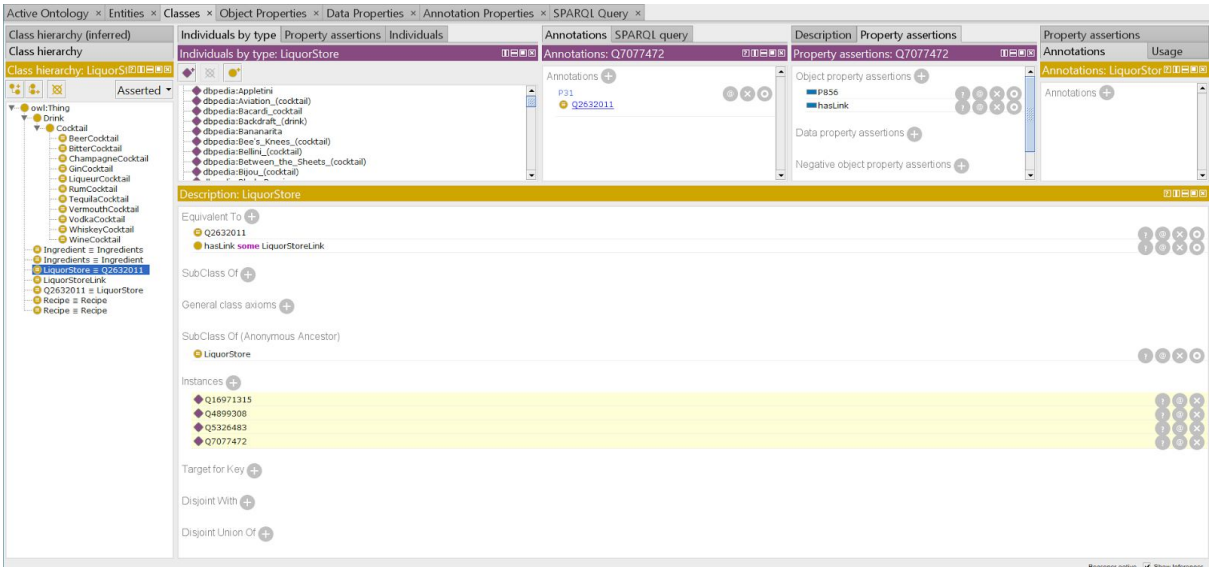
The class BeerCocktail is from our own ontology, the instances that are inferred are all DBPedia data. The other subclasses of Cocktail (all from our own database) also have instances all from DBPedia.



The Ingredients class is from our own ontology, the instances again from DBPedia. The instances are not very useful, what is important here are the rdfs:comments for all the ingredients: that is the actual useful part where the ingredients are described.



The Recipe class is from our own dataset, the inferred instances from DBPedia. Here again, the recipe names are useless, but for all the recipes there is a rdfs:comment as can be seen on the screenshot.



The LiquorStore class is from our own database, made equivalent to the Q2632011 class from WikiData. The instances are also from WikiData.

A higher resolution image of the screenshots is added with the other files with our submission.

MILESTONE 3 - Data reuse & Querying

External data sources
We used two external data sources:
1. DBPedia.
2. WikiData (SPARQL Endpoint).

All the different cocktails that we wanted to use are located in DBPedia. This data source also differentiated the cocktails by liquor type, which was exactly what we wanted. This data source also contained an ingredient list and a description on how to make the cocktail, for all the cocktails.
For the second data source, we used WikiData. This data source contained online liquor stores. We thought this was a good expansion for our application if we could also redirect the users to a liquor store where they could buy the products to make the cocktail of their choice.

Motivation data sources
DBPedia: We need this database since there are many cocktails available on DBPedia, and most of them have a property dbp:Ingredients and dbp:prep. This is perfect for our database, because we want to show for each cocktails the ingredients and a recipe. Another advantage of DBPedia was that the cocktails all have a property dbp:subject followed by the category of cocktails they are from: for example dbc:Cocktails_with_gin or dbc:Cocktails_with_wine. This was necessary since we want the user to be able to filter on the liquor used in the cocktails.
WikiData: We wanted some examples of shops where the user could buy their ingredients to make a cocktail in our ontology, so using those liquor stores from WikiData was ideal to extract those liquor stores.

Produce integrated data
We extracted data from both DBPedia and WikiData by using SPARQL CONSTRUCT queries, to return the data in RDF and saved both files in turtle format. Now we had to import those two files (the two external data sets) and our own dataset into one, integrated ontology. We started with a new, empty ontology in Protégé and imported all three files as a direct import. Now, we wanted to make the DBPedia data an instance of our classes that we created in our own data set (Cocktail, Ingredients and Recipe), and also the Wikidata data instances of the classes that we created in our own data set (LiquorStore and LiquorStoreLink). So the conceptualization that we created in Milestone 2 stays the same: we are not adding or deleting classes, we are only creating instances inside those classes. To make the steps with did clear, we explained it step by step.

Step 1: extracting DBPedia data
We extracted the data from DBPedia that we needed for our database with the following

query:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX db: <http://dbpedia.org/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX dbc: <http://dbpedia.org/resource/Category:>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX cl: <http://www.semanticweb.org/cocktailicious/>

CONSTRUCT {
?drink dct:subject dbc:Cocktails_with_liqueur.
?drink dbp:prep ?recipe.
?drink dbp:ingredients ?ingredients.
?drink rdf:type owl:NamedIndividual .
dbc:Cocktails_with_liqueur rdf:type owl:Class.
dct:subject rdf:type owl:ObjectProperty.
dbp:prep rdf:type owl:ObjectProperty.
dbp:ingredients rdf:type owl:ObjectProperty.

?recipe rdfs:comment ?recipe_str .
?recipe rdf:type cl:Recipe.
?ingredients rdfs:comment ?ingredients_str.
?ingredients rdf:type cl:Ingredient.
}
WHERE {
?drink dct:subject dbc:Cocktails_with_liqueur.
?drink dbp:prep ?recipe_str.
?drink dbp:ingredients ?ingredients_str.
BIND(URI(CONCAT("http://dbpedia.org/resource/recipe", MD5(STR(?recipe_str)))) as ?recipe)
BIND(URI(CONCAT("http://dbpedia.org/resource/ingredients", MD5(STR(?ingredients_str)))) as ?ingredients)
}
```

This gave us a file with all cocktails of a certain type, with a dbp:prep property following by a recipe: the recipe is converted to a string which is a rdfs:comment. Same for the dbp:ingredients property: the ingredient list is a rdfs:comment. In this example, we searched the dbc:Cocktails_with_liqueur, but we executed this query for all liquor types in our ontology.

Step 2: integrate DBPedia and own data

To infer that all the cocktails from DBPedia data are placed inside the right class, (so a cocktail containing gin must be inferred in the GinCocktail class and not in the VodkaCocktail class for example), we did the following steps:

We created for GinCocktail the Class Restriction

GinCocktail subject value Cocktails_with_gin. This inferred all the DBPedia cocktails containing gin inside the class GinCocktail of our own ontology. We did this for all the subclasses of Cocktail.

The next step is to place all the recipes (from the DBPedia data) that the cocktails have, in the Recipe class (of our own dataset). We did that by making the Recipe class that we created in our CONSTRUCT query (see: ?recipe rdf:type cl:Recipe), equivalent to the Recipe class from our own database (so they are both named recipe, but not the same, until we made them equivalent). After that, we gave the Recipe class the restriction *Recipe IsRecipeFor some Cocktail.*

We did the same thing for the ingredients inside the DBPedia data: we made the Ingredients class from the CONSTRUCT query equivalent to the Ingredients class that already exists in our own dataset. We gave that Ingredients class the class restriction *Ingredients areIngredientsFor some Cocktail.*
If we want to query our integrated data, we had to make sure that we could query for all cocktails for their recipe and their ingredients. To make that possible, we had to made some DBPedia properties equivalent to the properties of our own dataset: we made DBPedia property *dbp:prep* equivalent to our own property *hasRecipe* and *dbp:ingredients* equivalent to *hasIngredients.* Keep in mind that *hasRecipe* and *IsRecipeOf* are each others inverse, and so are *areIngredientsFor* and *hasIngredients* (see milestone 2). This made sure that each cl:Cocktail also has a cl:Recipe and a cl:Ingredients.

Step 3: extracting WikiData data
We extracted data from WikiData with the following query:

```
CONSTRUCT {?slijter wdt:P31 wd:Q2632011.
  wd:Q2632011 a owl:Class.
  ?slijter wdt:P856 ?link.
  ?slijter a wd:Q2632011.
  wdt:P856 a owl:ObjectProperty.
   ?slijter a owl:NamedIndividual.
   ?link a owl:NamedIndividual.
   }
  WHERE {?slijter wdt:P31 wd:Q2632011.
  ?slijter wdt:P856 ?link. }
```

This gave us a turtle file of all liquor stores, and for all liquor stores a website URL.

Step 4: integrate WikiData and own data
We wanted all the liquor stores in the WikiData file in the class LiquorStore (the class of our own database) and the website URL's of those stores in the class LiquorStoreLink of our own database.
Our class cl:LiquorStore has the class restriction *LiquorStore hasLink some LiquorStoreLink* and our class LiquorStoreLink has the class restriction *LiquorStoreLink isLinkTo some LiquorStore.*
The WikiData Class with the liquor stores is wd:Q2632011. We made class LiquorStore of our own database equivalent to wd:Q2632011. As a result, all the liquor stores are inferred in the LiquorStore class.
To get the URL's in the LiquorStoreLink class, we made wdt:856 (the WikiData property which has as object the website URL) equivalent to HasLink. Also, the properties cl:hasLink and IsLinkOf are each others inverse. This gave us all the URL's of the liquor stores inferred in the LiquorStoreLink class, the names of the LiquorStores inside LiquorStore and the names and URL's are connected with the hasLink and IsLinkOf properties.

SPARQL querying over integrated data

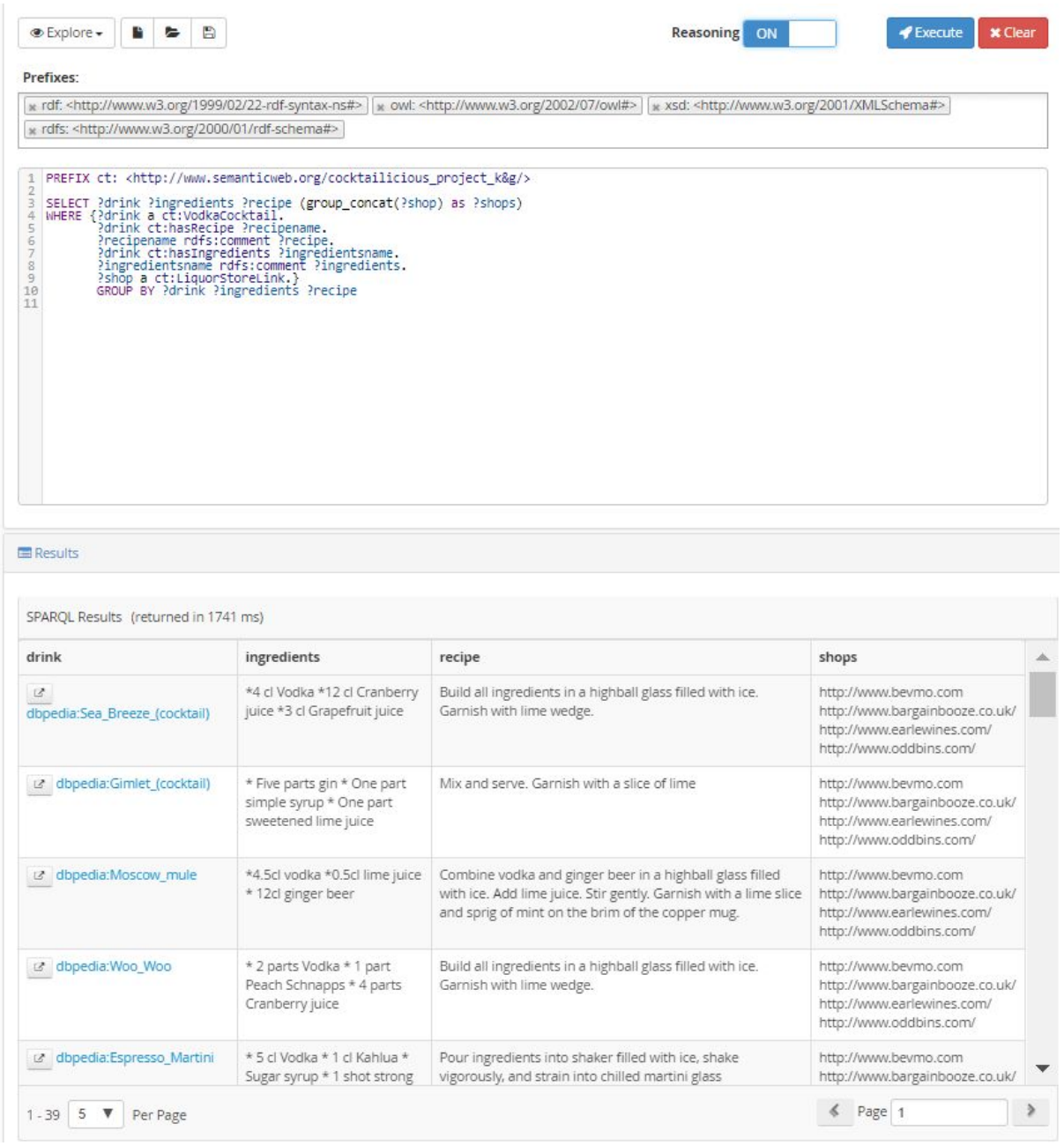
```
PREFIX cl: <http://www.semanticweb.org/ownontology_COCKTAILBASE/>

SELECT ?drink
WHERE {?drink a cl:TequilaCocktail.}
```

We first created this query, this is to retrieve all cocktails of a certain type. For our application however, we want also the recipe, ingredients and shops of the ingredients.

```
SELECT ?drink ?ingredients ?recipe (group_concat(?shop) as ?shops)
WHERE {?drink a cl:RumCocktail.
       ?drink cl:hasRecipe ?recipename.
       ?recipename rdfs:comment ?recipe.
       ?drink cl:hasIngredients ?ingredientsname.
       ?ingredientsname rdfs:comment ?ingredients.
       ?shop a cl:LiquorStoreLink.}
       GROUP BY ?drink ?ingredients ?recipe
```

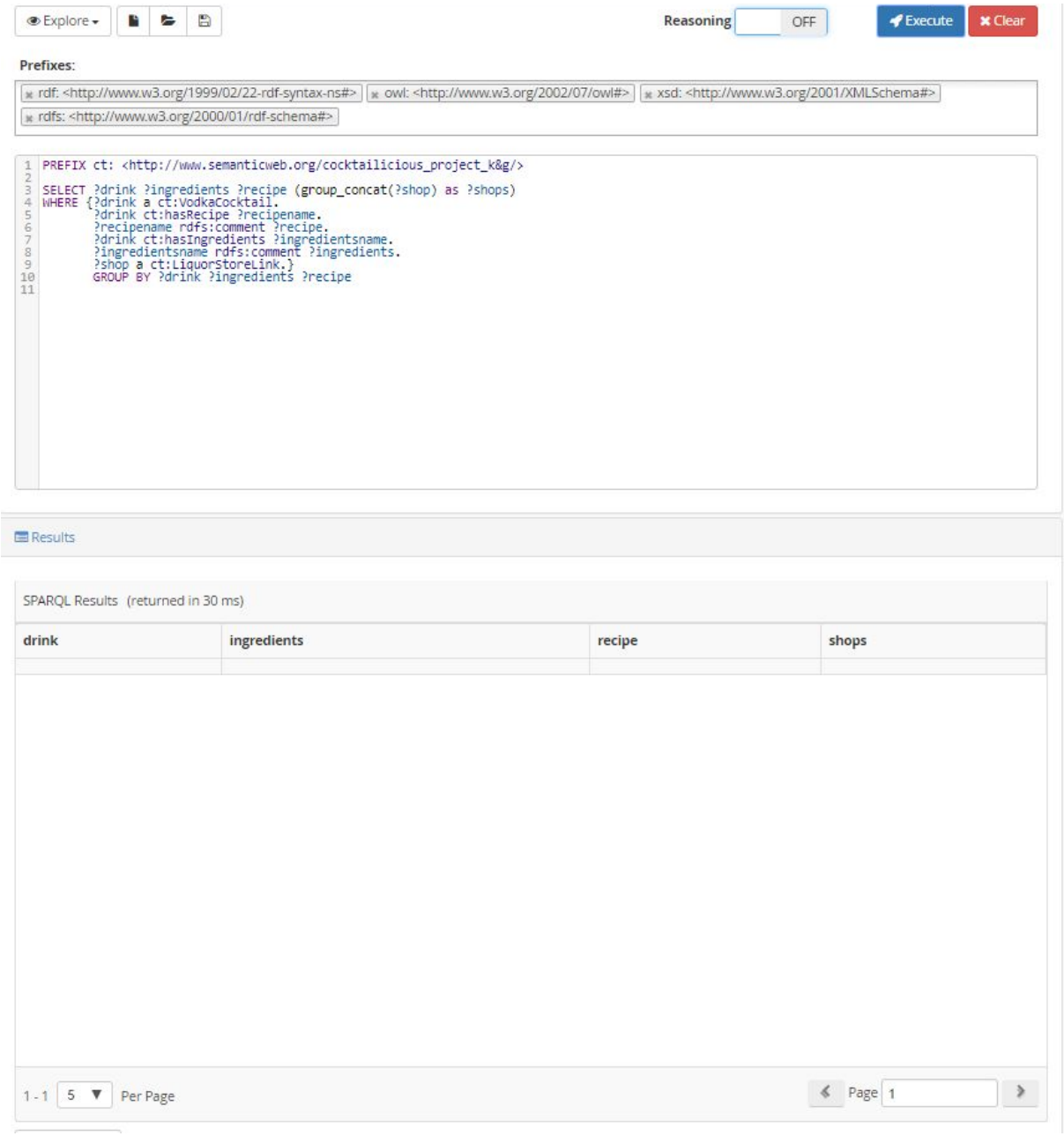
We want the user to choose a certain type of liquor, and then our application should show the user all the cocktails with that liquor as ingredient, and also a recipe of that cocktail, the ingredients of the cocktail and some liquor stores so that the user knows where to buy the ingredients. The ingredients and the recipe are inferred as instances of a certain cocktail type. The actual written recipe and ingredients (so not the WikiData name) are rdfs:comments of those instances.
So we query first for all cocktails with a certain liquor type, and then we query for the ?recipename of that cocktail and then also the ?ingredientsname. The user does not have to see those names, since they are just a bunch of numbers and letters and useless for the user. The user should see the comment, so we query for the rdfs:comment of the ?recipename and ?ingredientsname. Then, for all cocktails, we want to add all the liquor stores, so we add them with group_concat.

Screenshot with reasoning on:



When reasoning is on, everything shows up how we want for our application.

Screenshot with reasoning off:



So when reasoning is off, the SPARQL query return is empty. This makes sense, because we inferred all instances of all classes. So if the reasoner is off, Stardog does infer nothing and will therefore return nothing.

EVALUATION

We are not really satisfied with the result of our web application. All the queries that we created do work, but they are not nicely shown on our web application. We tried to fix this several times, but nothing seemed to work. We decided to accept it and realized that not everything can be perfect. Overall, we are happy that the main goals of the application do work (the database infers correctly, the queries return the correct results and the button

gives the results). The lay-out of the web application is just a detail that does not work how we wanted it to work. However, it is unfortunate that we could not fix it.