VRIJE UNIVERSITEIT BRUSSEL

PROJECT: FIRST RESPONSE

Open Information Systems

GROUP 6:
EMILE SONNEVELD 0546726
GREGORY NICOLAS WULLAERT 0526885
THIBAULT HENDRIECKX 0538150

Dec 21, 2018

Prof: Jan Arend Hidders

Wetenschappen en Bio-ingenieurswetenschappen

Abstract

This report describes our implementation for the Open Information System project 2018. We used the Django framework to build a website. On this website a user can manage his own medical record containing the symptoms he is experiencing and the diseases causing these symptoms. There are also doctors in the system, patients are able to contact these doctors. These doctors then can view this patient medical record and assist this patient were needed. Extra functionalities have been implemented as well. For example, doctors can add specialties to their profile and doctors can write articles. The patient has an overview page of all the articles that have been written about subjects relating to his medical record. Our team exists out of three people: Emile Sonneveld, Gregory Nicolas Wullaert and Thibault Hendrieckx.

Table of Contents

1 Introduction 2

2 Design choices 2

2.1 User stories relevant for the database design 2

2.1.1 General use-cases 2

2.1.2 Patient use-cases 3

2.1.3 Doctor use-cases 3

3 Database design 3

3.1 Entity-Relationship diagram 3

3.2 Resource Description Framework 5

3.2.1 RDF-Design 5

3.2.2 Database rules 6

4 SPARQL endpoint 6

4.1 Design 6

4.2 Example data 6

4.3 Example queries 7

4.3.1 Finding a disease based on symptoms 7

4.3.2 Listing all diseases in the database 7

5 Web-interface 8

5.1 Used technologies 8

5.2 Implemented functionalities 8

5.3 Graphical design 8

5.3.1 For patients 8

5.3.2 For doctors 8

5.3.3 The personalized articles 9

6 Conclusion 10

6.1 First Response project 10

6.2 Group and work division 10

References 11

1 Introduction

This project is a part for the course "Open Information Systems". The aim of this project is to design and implement an ontology-based web application. The subject of this application was an online doctors' practice. Which allows users to use it to get a medical diagnosis. The name of our project is inspired to the usage of the application. Patients are able to get a first diagnosis on their symptoms in just seconds and wherever they are. Only the second step is to contact a doctor. Therefore, this app is named "First response".

We give more information of our design choices in the 2nd chapter of this document. There we will give some explanation on general design choices as well as some concrete Use-Case scenarios. Further we elaborate on the database design itself in chapter 3. We illustrate this by using ER- and RDF-diagrams, also the database rules will be listed here. The assignment required the implementation of a SPARQL endpoint. The design and example queries can be found in chapter 4. Further, if end-users who want to use this system for the purpose of *getting a medical diagnosis*, they can use a web interface. This is presented in chapter 5. Lastly, we write our conclusions in chapter 6. Since this assignment had to be completed in groups of four randomly combined students, we will review our internal group work as well as a conclusion regarding our project in general.

2 Design choices

The first aspect of our database is building a design which allows us to store 'Symptoms' and 'groups of symptoms'. This is done by storing them in the same table, which is called 'Symptoms'. Symptoms can have sub-symptoms and symptoms can inflict other symptoms. This is modelled with 2 self-referring relations. The first self-reference is because it is not clear whether symptoms are just symptoms or a group of symptoms. E.g. Dizziness is in the group of neurological symptoms. But dizziness itself can be a group that contains 'light dizziness'. The other self-reference is because symptoms can inflict other symptoms this is marked in the ER-model with a 'can cause'-relationship.

Doctors and patients have accounts. They are separated with a normal ISA-relationship. In other words, one account can be a doctor or a patient. Patients need to provide their birth date, as many diseases are age related, and doctors may provide a specialty, this can be a specialty in diseases written in plain text. Because every user needs to be contactable, we make the email address required, and use it as primary key in the user table. Changing email address is not possible in our application. The implementation defines to what access users have access to in the database.

As required, the primary keys to refer to a disease or symptom should be human readable. That is why we choose to take their initial name and generate a slug from it. A slug is human and machine readable, so we can use it as primary key.

2.1 User stories relevant for the database design

There are so many use-cases we can write down for this application. To keep this report to the point and interesting, we will only write down the most important use-cases which are required for the assignment. In general, these use-case are relevant for the database design or specific to our application in contrary to some more generic use-cases for every application like being able to visit the website or clicking specific buttons.

2.1.1 General use-cases

Every user on the website is able to <u>browse through all symptoms</u> in the database. For the symptoms they've selected they are able to <u>find a relevant disease</u>. Every user, patient or doctor, is able to <u>contact a doctor</u> who is specialised in a certain domain.

2.1.2 Patient use-cases

First of all, every new patient needs to be able to register in the application. Upon registering, he will be able to provide a name and password and also every user is able to provide a birthday. When the patient has registered and became a user of the system, he is able to add and remove symptoms to/from his profile. Based on these symptoms he receives a first diagnosis. To every symptom the user can add a severity indicating the how *hard* he experiences the symptom. The user is able to send his medical record to a doctor. In our more complex system. The user can also read about his own condition in the articles written by doctors on the website.

2.1.3 Doctor use-cases

Doctor profiles differ from regular user profiles in their registration. Their account can only be added to the system by the doctor's practice manager (system admin). Doctors do can change their profile. They are able to add specialisations to their profile. Which are relevant for the articles they write for their patients. In regards of patients, doctors are able to view patients' medical records including symptoms a patient removed from their profile. They can also alter the database by adding/removing symptoms. And they are able to contact patients.

3 Database design

3.1 Entity-Relationship diagram

The Entity-Relationship (ER) Diagram show the interconnection between different data objects. This is how our real relational database was formed. When exposing our relational database as RDF, we want to be able to have a relation to the DOID. That's why in our relational database, we extend Symptom and Disease with an extra field: '*uri*'. Here we store a reference to the well-known Symptom and Disease ontology. The 'has' relationship between Patient and Symptom gets a separate RDF class, called Diagnose. This way, we can add properties to this relation (*severity*, *start_date* and *end_date*). When a user removes a symptom from his profile, it is disregarded from the system to provide a diagnosis and also made invisible for the user. We keep this information in our database so the doctor can have a history on the user's medical record. This is useful for detecting chronic diseases.

We added more complexity to the system by adding the ability for doctors to write *blog*-like articles on the platform. These articles are then tagged to symptoms. When a user has some symptoms actively added to their profile, we can match those symptoms to the tags of the articles showing a personalized medical news feed in the application. This allows users to be ever updated on new developments on his/her disease.

Symptoms and diseases have a slug-field. This is the primary key for those objects. These field are not user-generated though. These objects have a regular name field containing all sorts of characters. The slug field will remove most exotic characters leaving the remaining string a human readable string but only containing characters from a strict alphabet. As an example, "Tourette's Disease" will be transformed to "*tourettes_dis*ease*".*

The new conceptual schema for the relational database, including the feedback from intermediate assignments, is shown in figure 1.

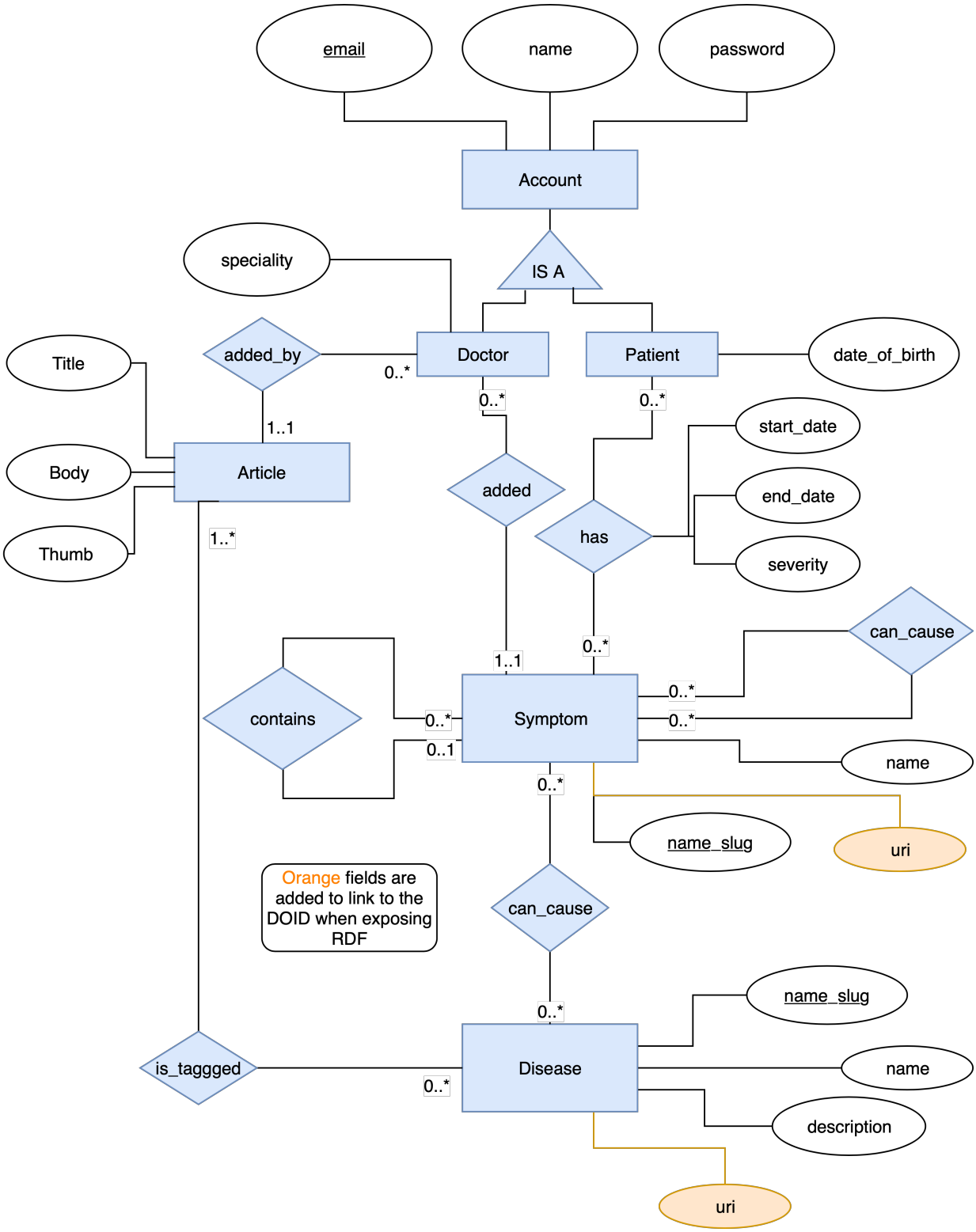

Figure 1: ER-model for our application

3.2 Resource Description Framework

3.2.1 RDF-Design

The Resource Definition Framework (RDF) design of our application is built upon our ER-model. Here we can see all the same information again. Also, here we added more complexity as feedback on the intermediate assignments.

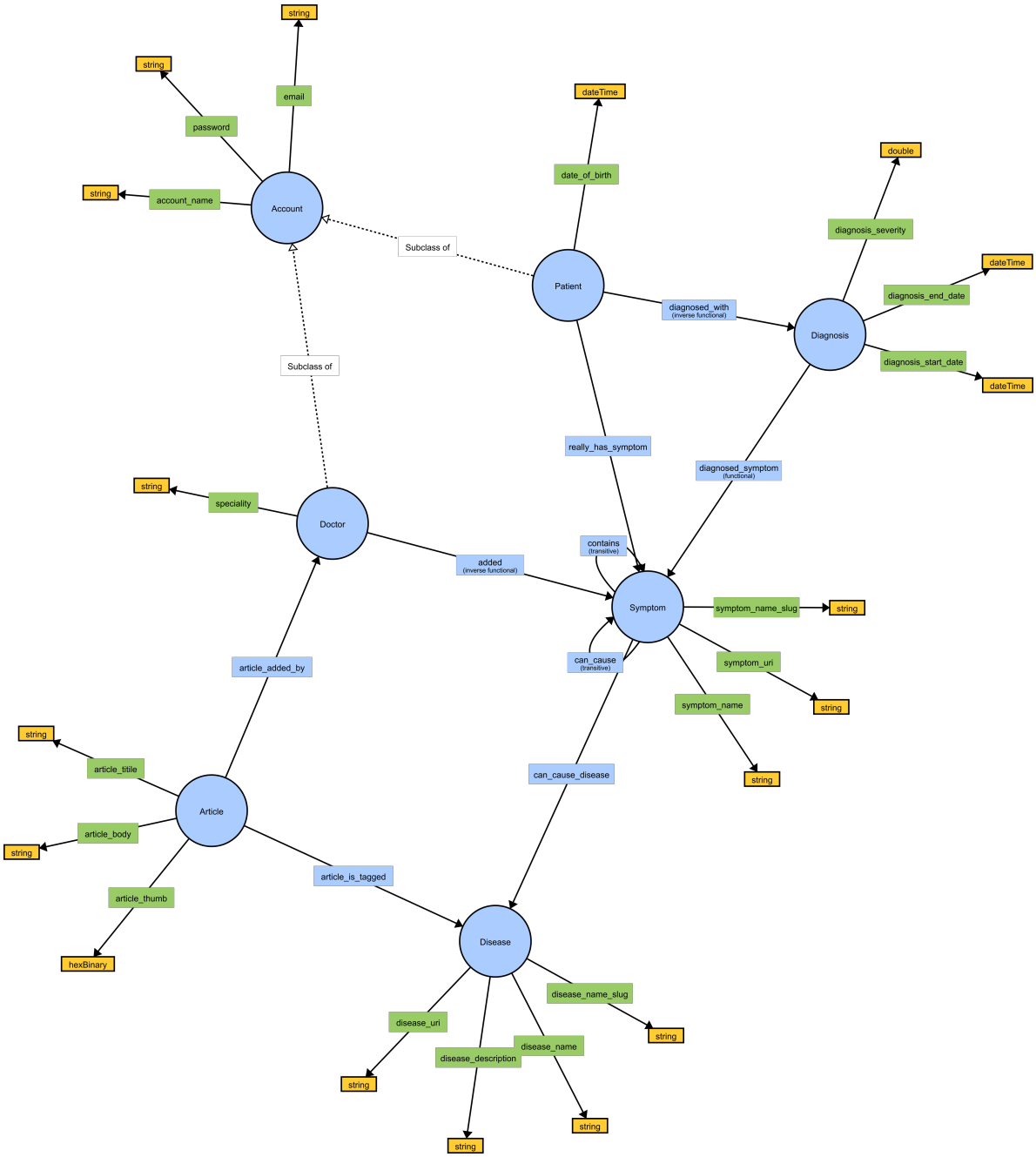

Figure 2: Database RDF design

3.2.2 Database rules

We've added 2 rules to the ontology:

- When a patient has been diagnosed with a severity more than 5, we add a relation from the Patient directly to the Symptom; "really_has_symptom". This is because some clients of our RDF could have no recognition of symptom severity. We implemented this as a fixed cut-off. Meaning when the severity is below a 5 out of 10, we will not match this upon a disease.
- When symptom A contains symptom B and symptom B contains symptom C, then symptom A will contain symptom C. This is done to provide a cleaner symptom tree. We did change one element in the database to allow this to happen smoothly. We added a root symptom. This symptom contains all top-level symptoms directly. And thus, by this rule, contains all other symptoms indirectly as well. This is implementing by keeping the tree structure but to allow easier matching of relevant symptoms.

We tried to add a rule to indicate when all the symptoms linked to a disease are also linked to a patient, this patient has this disease. But due to limitations in OWL, we did not manage to express this. So, we considered generating a set of rules, one per disease. But *Ontop* does not allow generating rules (Cornam, 2017).

4 SPARQL endpoint

4.1 Design

We've implemented our SPARQL endpoint so it's accessible from the web browser. Here it is possible to query live data from our database. However, the database is also accessible with the Protégé plugin. The plugin is configured to use our remote PostgreSQL server. This generates an owl-file. Furthermore, we use the HermiT reasoner to calculate the RDF-Rules. Protégés Ontop plugin handles the mapping between our relational database and RDF. This is via an obda-file, which contains all the settings to connect to the local database. The live environment runs Apache Tomcat, this is because it supports the Ontop Virtual RDF Store (Ontop, 2018).

4.2 Example data

In the first iteration we've added example data in our RDF. A sample from that example data is shown in figure 3. From this figure we can read that '*Johnny Doe*' has the '*dancing_plague_of_1518*' disease. As explained in chapter 4.1 this data is not the real data in the database anymore, but the data relationship resembles our real database.

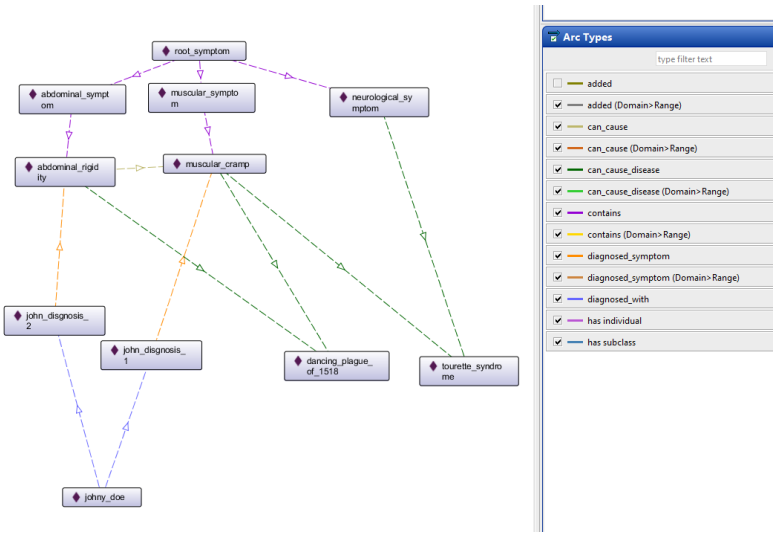

Figure 3: Example data in RDF format

4.3 Example queries

To be able to run all our queries we need to set these prefixes. Hereafter we can run our two example queries (4.3.1 & 4.3.2)

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX obo-term: <http://purl.obolibrary.org/obo/>
PREFIX oboInOwl: <http://www.geneontology.org/formats/oboInOwl#>
PREFIX g6: <https://www.vub.ac.be/group-6#>
PREFIX obo: <http://purl.obolibrary.org/obo/>
```

4.3.1 Finding a disease based on symptoms

In this query we start from a situation where we have two symptoms occurring. This means for our system the severity the patient gave these symptoms is higher than 5 out of 10 (see chapter 3.2.2). This query would read in human language as "*Select all diseases that cause a muscular cramp together with neurological symptom*".

```
SELECT * WHERE {
  g6:muscular_cramp g6:can_cause_disease ?b.
  g6:neurological_symptom g6:can_cause_disease ?b.
}
```

The SPARQL-endpoint gives this result. Which again in human language means that when a patient is having muscular cramps while he is known to have a neurological symptom, He probably suffers from Tourette's Syndrome.

```
g6:tourette_syndrome
```

4.3.2 Listing all diseases in the database

In this query we will make a list of symptoms and list their OBO label. This is useful when a user would like to know which symptoms exist or to get more information on some symptoms. The human readable version of this query is "*Select all symptoms and show their OBO label*".

```
SELECT * WHERE {
  ?symp g6:symptom_uri ?url.
  ?oboSymp oboInOwl:id ?url.
  #?oboSymp obo:IAO_0000115 ?description.
  ?oboSymp rdfs:label ?label.
}
```

The SPARQL-endpoint gives this result (shortened). Which reads in a table format for every symptom: a slug, a uri (to external database), the OBO and a human label.

	symp	url	oboSymp	label
1	g6:tourette_syndrome	"DOID:11119"	obo-term:DOID_11119	"Gilles de la Tourette syndrome"
2	g6:abdominal_rigidity	"SYMP:0000454"	obo-term:SYMP_0000454	"abdominal rigidity"
3	g6:abdominal_symptom	"SYMP:0000454"	obo-term:SYMP_0000454	"abdominal rigidity"
4	g6:muscular_cramp	"SYMP:0000093"	obo-term:SYMP_0000093	"muscle cramp"

Table 1: Results from example query showing OBO-labels

5 Web-interface

5.1 Used technologies

For our web-interface we've chosen the Django framework. We did this because this framework provides a clean and extensive python model to work with relational data. Django allows to choose a database backend. Since we encountered some limitations with the SQLite database, we chose to migrate to a Postgres backend. Django also allows us to make easier user sessions and we don't need to worry about some basic implementations of the web framework. Lastly our choice of Django was supported by their advanced templating engine.

5.2 Implemented functionalities

In chapter 2.1 we discussed some of the use cases our system supports. All of these use cases have been implemented. In chapter 5.3 we will present a demo of the website using some screenshots. Further we made the SPARQL endpoint a part of the web interface. This allows other developers to access our database or user who have a limited knowledge of SPARQL can try queries out online. This means our web interface consists of two parts. The website where users have a simple interface where they provide the same information as they would provide to their doctor. Only when they would feel this to be insufficient or they would need some prescription drugs, a patient can contact a doctor who could threat the patient's medical record as a second opinion.

5.3 Graphical design

5.3.1 For patients

In figure 4 we see two pages available to the patients. In the left page a user can select a symptom to add to his/her profile. This is a user searchable table where a user can select a symptom based on the symptom's name. On the right-side page there is the overview a user gets when he accesses his/her medical record. For each symptom he submitted there is a start date and, end date and a severity. Based on the symptoms and the severity of the symptoms, the systems predicts diseases in the "probable diseases" section. The user then can click on the DOID number to get additional information on his/her disease. Also, it may be that a disease inflicts another disease. This allows the user to view other possible diseases. Below that there is a button that a user can click to contact a doctor. The doctor is automatically chosen by the disease a user has. Communication between the user and the doctor is done by email.

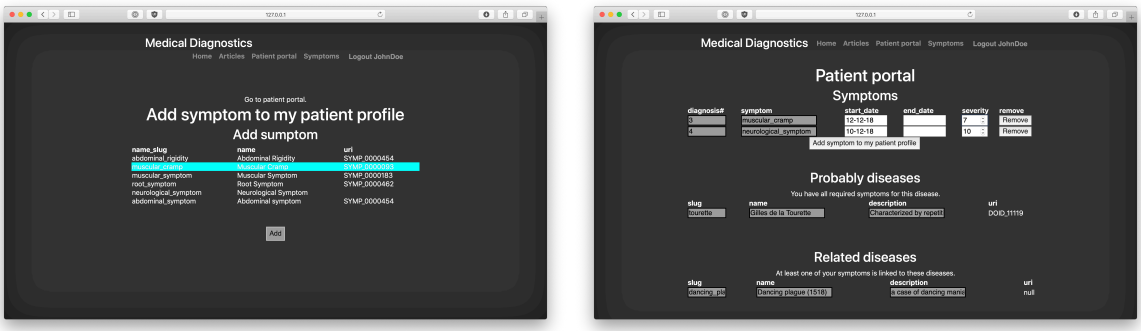

Figure 4: An overview of a patient's medical record

5.3.2 For doctors

Doctors can access different pages than patients. For example, a doctor can add symptoms to the system. In figure 5 two pages are shown that a doctor can use. The first page shows how a doctor can add a single symptom. He can enter all the necessary information for a patient to find this symptom. He can also provide a disease which is inflicts this symptom.

On the right-hand side, a doctor has an overview of all the symptoms he added in this system. It also shows symptoms added by other doctors in the same practice.

Figure 5: An overview of the symptom pages for a doctor

5.3.3 The personalized articles

We discussed the additional feature of this system, personalized articles, back in chapters 2.1.2, 2.1.3 and 3.1. In figure 6 these pages are shown. The left page is a page viewable by the patients. Here they will see a filtered list of articles, written by any doctor, but limited only to the diseases they currently have. These diseases are automatically retrieved from their medical record. Doctors write these articles themselves. These articles are tagged with diseases. The doctor can overwrite these himself, but this automatically matches by his specialty.

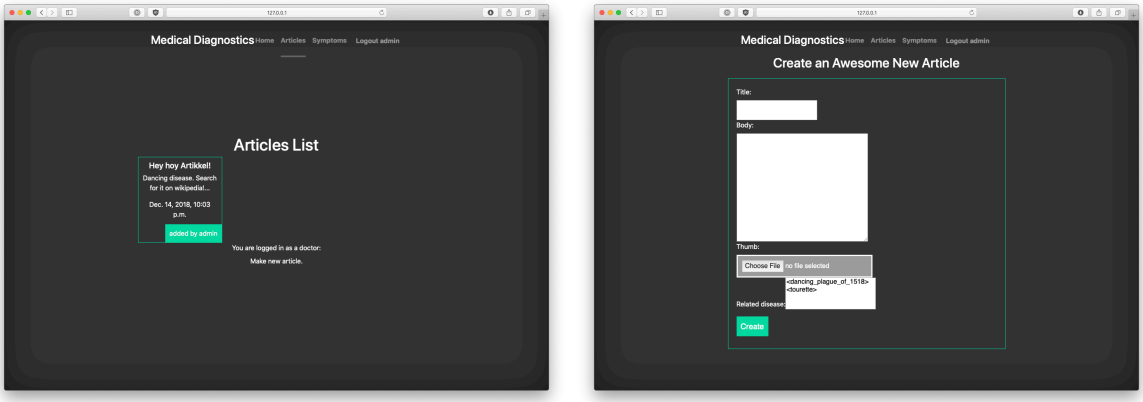

Figure 6: Pages showing the personalized articles

6 Conclusion

6.1 First Response project

First Response is how we branded our project. We sort of feel we succeeded in building this project. What went well was how we managed to get to every deadline, only by the last deadline we ran short on time (read: short on people). In the beginning there was a lot of uncertainty about what the specific requirement for the project were. We made our own interpretation of the project and we tried to add complexity at every deadline. The feedback we got was constructive and we used this for every new iteration. Also help from the assistant was always available.

We did want to add even more features. We tried to show a cure for diseases automatically added when the system diagnosed a patient. And a paywall, since doctors also receive honorary, which could be replaced by our system. We would have implemented a dummy paywall of €25 per diagnosis. This would be implemented as a 24hour free-pass to our system. Of course, via our SPARQL endpoint this would remain free. Medical books in the library are also free as well.

6.2 Group and work division

For this section we will write this conclusion together as Emile, Gregory and Thibault since we all feel the same about this. Our group was randomly set together. Which was an experience for all of us. Nobody knew each other before. Which turned all out fine in fact. We got along directly. We were able to start right away, everybody filled in his own role and has his own excellence. Specifically, we did the ER-diagram, preparation of presentations, and high-level design together. In the end, Thibault wrote the report, Emile did everything SPARQL related and Gregory handled the website. We did manage to get every intermediate deadline. Only at the end Vanja decided to drop this course. This led us to a situation where we had to get her work over. Due to a heavy workload from other courses which all also came in only 2 weeks before this deadline we had to pull a real extra effort to replace Vanja. In the end we believe we still managed to get a viable product leaving only a few places unfinished.

References

Cornam, J. (2017, August 8). *TurtleSyntax*. Retrieved from Github: https://github.com/ontop/ontop/wiki/Turtlesyntax#mapping-restrictions

Ontop. (2018, 12 20). *Github/Ontop.* Retrieved from https://github.com/ontop/ontop/wiki/RDF4J-SPARQL-endpoint-installation#installing-ontop-using-the-prepacked-rdf4j-workbench