UNIVERSITY OF PISA

Large Scale and Multi-Structured Databases

Year 2020/21

VisitEasy

ROSSELLA DE DOMINICIS

GIUSEPPE MARTINO

DARIO GABRIELE

INDEX

1 INTRODUCTION 1

2 REQUIREMENTS ANALYSIS 2

2.1 Application Actors 2

2.2 Functional Requirements 2

2.3 Non-functional requirements 3

2.4 UML Class Diagram 3

2.5 Use-Case Diagram 4

3 DATABASE ORGANIZATION 6

3.1 Dataset organization 6

3.2 MONGODB 7

3.2.1 Entities handled 7

3.2.2 Queries Handled 7

3.2.3 Collection structure 7

3.2.4 Velocity of data 11

3.2.5 Indexes 11

3.2.6 Indexes performances 13

3.3 GRAPH DB 16

3.3.1 Nodes 16

3.3.2 Relationships 16

3.3.3 Graph structure 16

3.3.4 Queries handled 17

3.3.5 Indexes 18

4 DATABASE PROPRIERTIES 20

4.1 Redundancies and reservations handling 20

4.2 Consistency, Availability, Partition Tolerance 20

4.2.1 Cross Database consistency 22
5 IMPLEMENTATION 23
5.1 Package structure 23
5.1.1 Entities 23
5.1.2 Persistance 23
5.1.3 Utils 24
5.2 Project Object Model (POM) File 24
5.3 Analytics and Aggregations in MongoDB 25
5.4 Cypher implementation queries in Neo4j 27
Add a review. Create nodes of doctor and user if they don't exist: 27
5.5 Technologies and Frameworks 28

1 INTRODUCTION

VisitEasy is a Java application that allows people to find doctors by cities and specializations, book medical examinations with them, write reviews about doctors and read other users' reviews.

Once the user is on the homepage, he is invited to sign up if he is not registered yet, otherwise, he can log in using his username and password.

An user can easily view a list of specializations and cities of the doctors present on the app and choose them. All the related doctors will be displayed. To help users to choose a doctor, he can also see a list of "best doctors" for each city and specialization, based on reviews that other users wrote.

Indeed, when a user selects a doctor, he can see other people reviews about that doctor, put a "like" on a review that he believes helpful and can write himself a review.

The user can also see the doctor's profile, which shows some information like his address, the price of the medical examination, and the biography of the doctor.

He can also book a medical examination, choosing among all available slots and see all the history of his own reservations of medical examinations.

As for the user, also a doctor can access the app. On his homepage he can see all the bookings on his calendar, according to the time slots that he had made available.

He can see all the reviews that people wrote about him and he can modify his information like the price of the medical examination, his address and his biography.

The administrator of the app is in charge to guarantee that all users must not write disrespectful posts, so he can delete reviews that he considers inappropriate, and he can also ban users and doctors from the application.

The administrator can also see some statistics about the usage of the app.

2 REQUIREMENTS ANALYSIS

2.1 Application Actors

The actors of the application are the *User*, the *Administrator* and the *Doctor.*

Users and *Doctors* are those who make direct use of the app.

Administrators have additional functionalities to manage the app in the best way.

2.2 Functional Requirements

The *functional* requirements of this application, divided concerning the three actors, are as follows:

- The application is available only for registered users and registered doctors.
- A *User* can browse the list of the doctors related to a city and a specialization.
- A *User* can see the list of "best doctors" of a city and a specialization, recommended by the application based on the reviews of other people.
- A *User* can view a doctor profile.
- A *User* can view all the reviews about a doctor.
- A *User* can book a medical examination for an available date and time.
- A *User* can add a review on a doctor.
- A *User* can view the list of all his own reservations.
- A *User* can delete a reservation.
- A *User* can put a "like" to a review that he believes helpful.

- A *Doctor* can modify information on his homepage (price, address and biography)
- A *Doctor* can view all his reviews made by users.
- A *Doctor* can add new available slots on his list of reservations.
- A *Doctor* can view all his reservations.

- An *Administrator* can delete reviews if he considers them offensive.
- An *Administrator* can add another administrator.
- An *Administrator* can add a doctor.
- An *Administrator* can add a user.
- An *Administrator* can delete a doctor.
- An *Administrator* can delete a user.
- An *Administrator* can see the list of cities where the app is more used by users.
- An *Administrator* can see the list of most expensive specializations.

- An *Administrator* can see the list of most active reviewers.
- An *Administrator* can see the list of best reviewers.

2.3 Non-functional requirements

- The application should guarantee data **availability**.
- It should guarantee an **eventual consistency** (*read your write, monotonic read, monotonic write*) in the whole process of booking a medical examination: after a user books a medical examination, he must see that this slot is booked by him and when he sees it in his list, no other users are allowed to see that slot like available. Moreover, several updates on a reservation made by a user must be executed in order.
- The operation of booking or deleting a medical examination must be **atomic**: a reservation present in the doctor's list must be present also in the user's one.
- The system must periodically update list of reservations of users and doctors, deleting old reservations.
- The application should be easy to use.
- The application will store information in non-relational Databases

2.4 UML Class Diagram

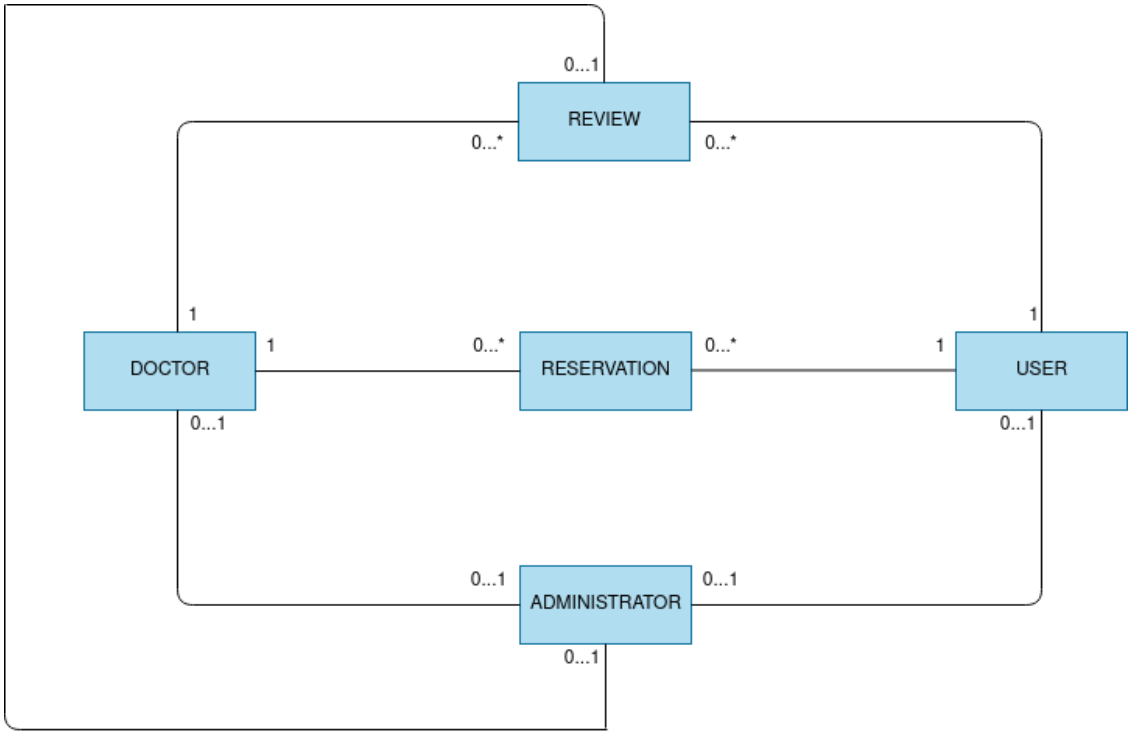

Figure 1: CLASS Diagram

2.5 Use-Case Diagram

In the picture we can see the Use-Case diagram.

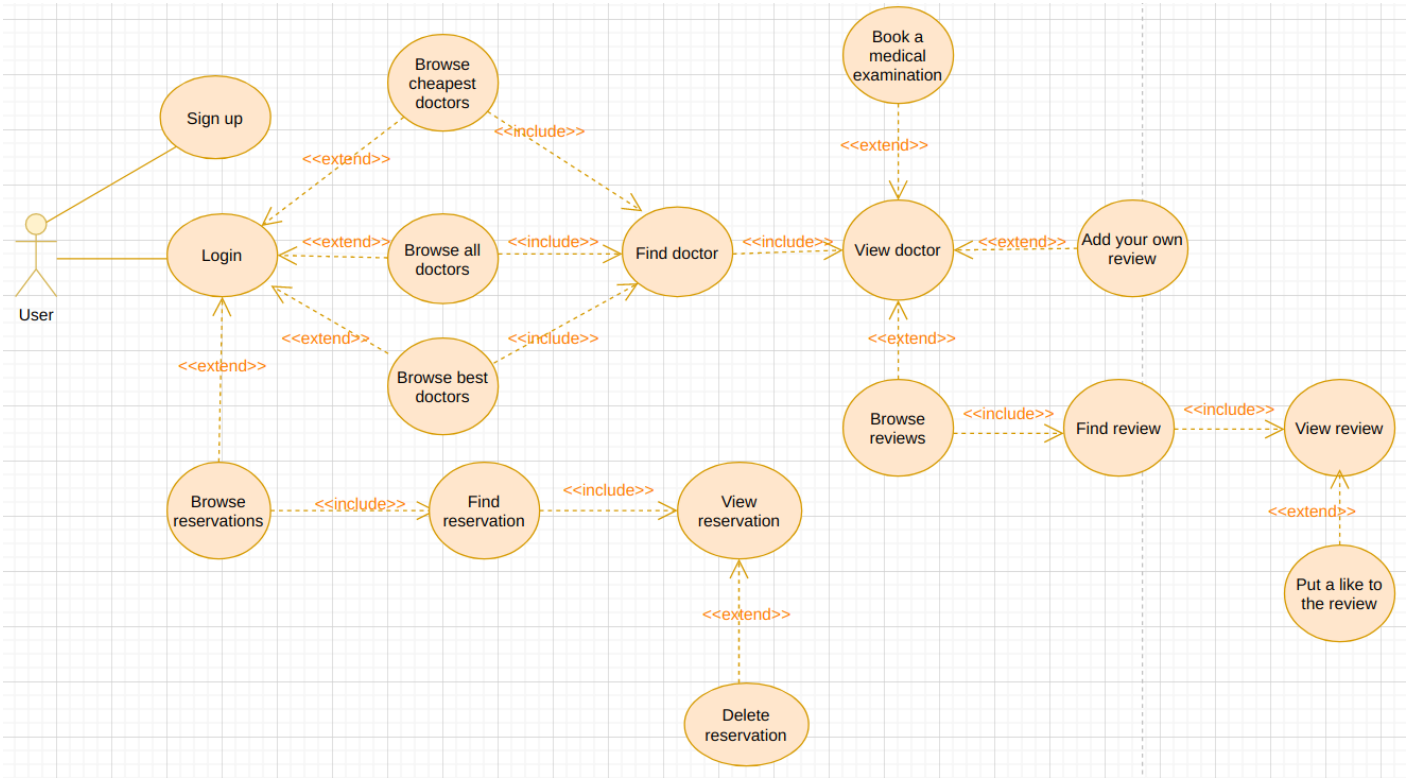

Figure 2: Use Case Diagram - User

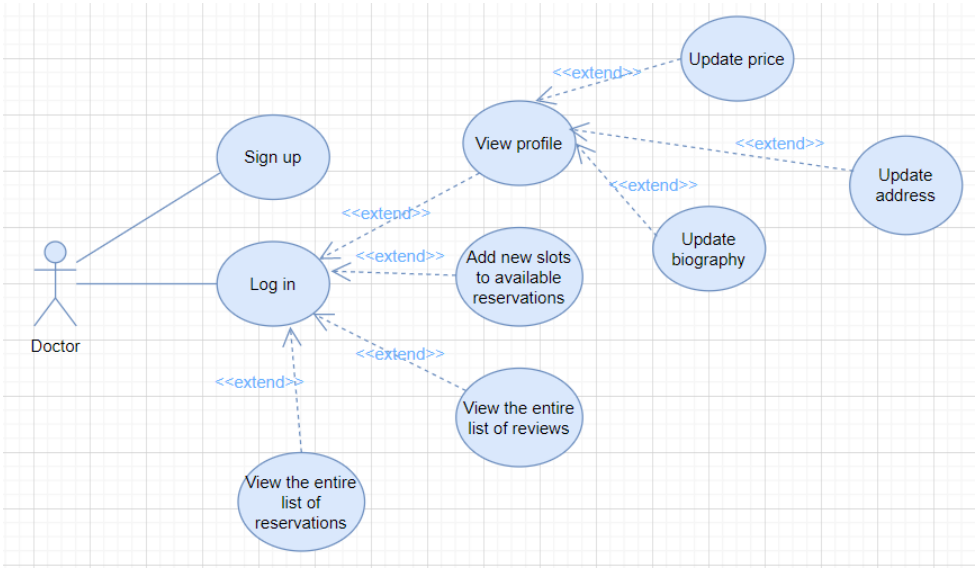

Figure 3: Use Case Diagram - Doctor

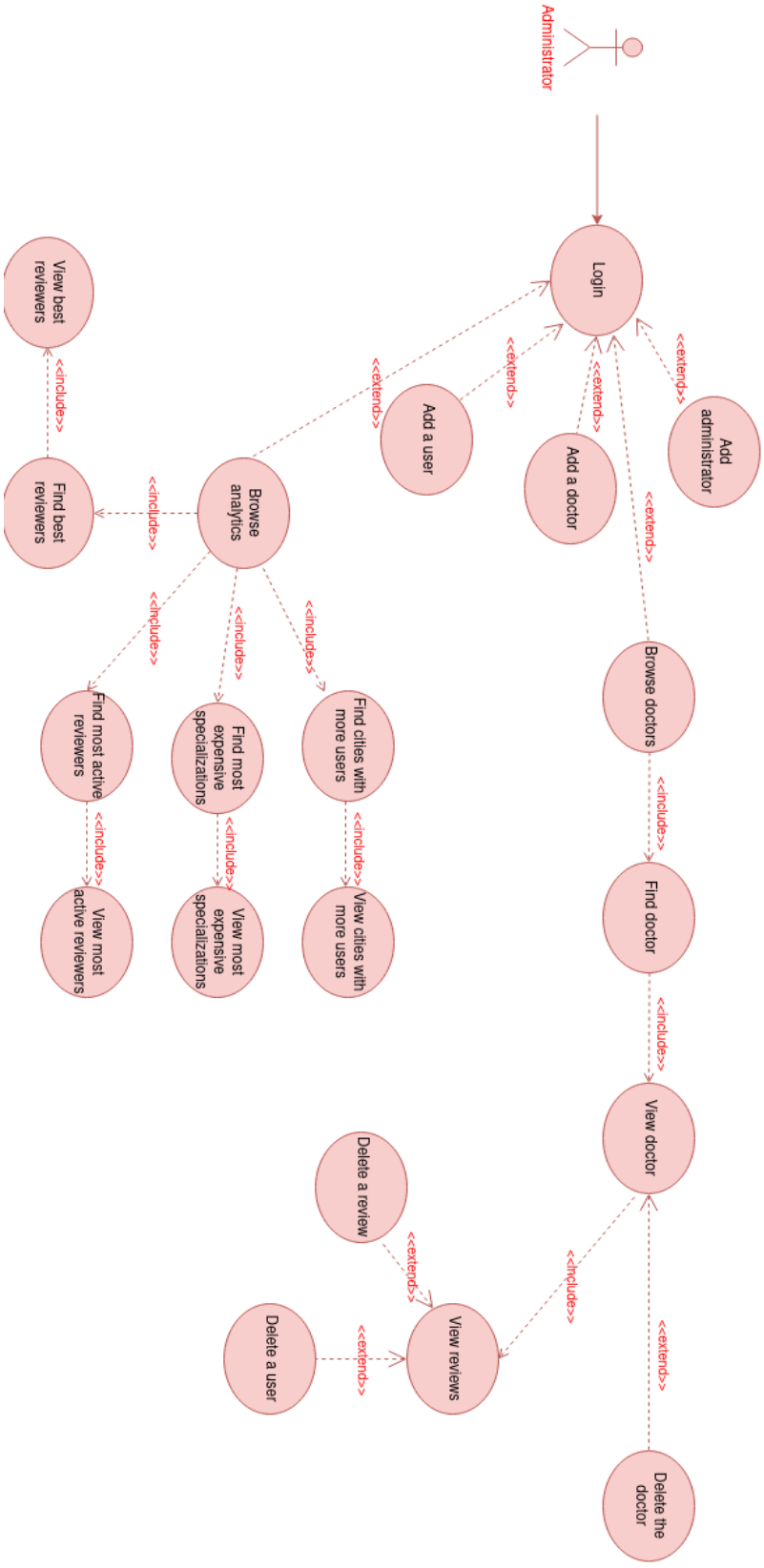

Figure 4: Use case Digram- Administrator

3 DATABASE ORGANIZATION

We decided to use two different DB to store all information needed:

1. A **Document Database**, using **MongoDB** for handling registration and authentication of doctors and users, all the booking of medical examination process and all information about doctors.
2. A **Graph Database,** using **Neo4j,** for handling the "social" part of the application based on doctors' reviews written by users.

During this chapter will be explained the more detailed motivation of these choices, and all the information about the two databases.

3.1 Dataset organization

To populate our databases, we performed a web scraping from different websites where are present information and reviews about doctors.

In particular we used the tool *ParseHub* to scrape data from the websites *MioDottore.it, iDoctors.it* and *Yelp.com*

Within the scraping we obtained data about doctors (name, city, specialization, price, address, biography) and their reviews (text, rating, data of the review, username of users).

Data about doctors and reviews are real.

We needed to modify raw data obtained from the scraping in order to satisfy the application needs.

More detailed information about how we modified data are present below in the paragraph of MongoDB and Neo4j.

After the preprocessing, the amount of data added in both databases weights 86 MB.

3.2 MONGODB

3.2.1 Entities handled

Considering the UML Class Diagram illustrated above, the entities handled by the Document DB are:

- **Doctor**
- **User**
- **Reservation**
- **Administrator**

3.2.2 Queries Handled

The implementation of the main queries and aggregations will be presented in the dedicated paragraph

- Insert a new Doctor
- Insert a new User
- Retrieve Doctor's username and password at login time
- Retrieve User's username and password at login time
- Retrieve Doctor information
- Retrieve all cities of doctors present in the DB
- Retrieve all specializations of doctors present in the DB
- Retrieve all doctors names and usernames of givens cities and specializations
- Retrieve all doctors names and usernames
- Update Doctors information (address, price, biography)
- Delete a Doctor (Administrator only)
- Delete a User (Administrator only)
- Add a new Administrator (Administrator only)
- Add new free slots (new empty reservations) (Doctor only)
- Add new reservation (User only)
- Delete a reservation
- Delete a group of reservation for each doctor given a range of dates
- See own reservations (Doctor and User)
- Analytics: see ranking of cheapest doctor given a city and a specialization (User only)
- Analytics: show city with most users (Administrator only)
- Analytics: show most expensive specializations (Administrator only)

3.2.3 Collection structure

In the document DB we stored data in 3 different collections:

- **Doctors**
- **Users**
- **Administrator**

After the scraping, we had a JSON file where each document was in this format:

Figure 5: JSON document

We add randomly password to all doctors in the doctor collection directly with MongoDB code:

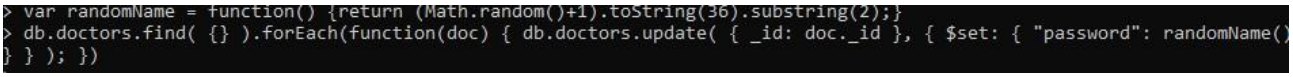

Figure 6: function password

Moreover, we managed some data types because, in the scraping we obtained every field in a string format, but to compute the analytics operations described in the functional requirements section, we needed to have some field in a integer format.

Here an example of query to convert the field "price":

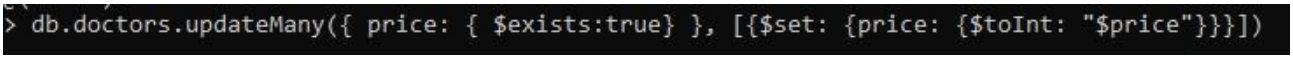

Figure 7: function price

Due to privacy issues and the consequent difficulty to retrieve data on the web about medical examination reservations, we created for each doctor an array of "reservation", where a single reservation is a document composed by the field "date" which represents a time slot where a single patient can book a medical examination and another field "patient" with the username of the user who booked the examination.

We assume that, when the field "patient" is empty, it means that there is no patient for that reservation, so the timeslot is free and available to any patient who wants to book a medical examination with that doctor in that time slot.

At first all reservations were empty, so we used our app to populate the "reservation" field making some random bookings.

This is the java code we used to insert the array of empty reservations to each doctor:

```java
public static List<DateTime> getDateRange(DateTime start, DateTime end)
{
    List<DateTime> ret = new ArrayList<DateTime>();
    DateTime tmp = start;
    while (tmp.isBefore(end) || tmp.equals(end)) {
        ret.add(tmp);
        tmp = tmp.plusDays(1);
    }
    return ret;
}
```

Figure 8: getDateRange function

```java
void aggiungi_cal7() {
    ArrayList<String> ore = new ArrayList<>();
    ore.add("09:00");
    ore.add("11:00");
    ore.add("15:00");
    ore.add("16:30");

    DateTime start = DateTime.now();
    DateTime end = start.plusMonths(2);

    List<DateTime> between = getDateRange(start, end);

    for (DateTime d : between) {
        for (String o : ore) {
            Document newres = new Document("date", d.toString(DateTimeFormat.shortDate()) + " " + o).append("patient", "");
            doctors.updateMany(new Document(), Updates.push( fieldName: "reservations", newres));
        }
    }

}
```

Figure 9: add calendar function

We deleted the field "review" because we decided to store reviews' data only in Neo4j.
At the end the collection "**Doctors**" results composed by documents like this:

```
_id: ObjectId("5feb218e3a65753d7c4b4917")
name: "Dott.ssa Maria Flavia Mazza"
price: 40
address: "Via del Torchio 3"
bio: "Dopo aver conseguito la laurea in "Scienze Biologiche" ho deciso di in..."
specialization: "Nutritionist"
city: "Milano"
password: "ct3xbqnevof"
username: "5feb218e3a65753d7c4b4917"
reservations: Array
  0: Object
      date: "23/01/21 09:30"
      patient: "AS"
  1: Object
      date: "23/01/21 11:30"
      patient: ""
```

Figure 10: document Doctor

We created the collection **Users** getting data from the usernames of doctors' reviews because they need to be the same, in order to have real result with query about reviews; then we added city and password randomly.

We decided to store reservations also in the collection User, so if the user makes some reservation, each document will have an additional array of documents called "reservations", where each document is composed by the field "date", which represents the time slot of the reservation, the field "doctor", representing the username of the doctor, to identify him uniquely, and the name of the doctor.

There is an example of a document in the collection Users:

Figure 11: Document User

The collection **Administrators** is composed by 2 fields: username, password. Only an administrator can add a new administrator, and there is a check if the username already exist.

There is an example of the document in the collections Administratorss:

Figure 12: Document Administrator

3.2.4 Velocity of data

The velocity is guaranteed by the dynamic reservations mechanism: every day the system deletes the old dates starting from the current date.

Method that deletes reservations every day from the Doctors and Users collection:

```
void deleteDate() {

    ArrayList<String> orari = new ArrayList<>();
    orari.add(" 09:00"); orari.add(" 09:30"); orari.add(" 10:00"); orari.add(" 10:30");
    orari.add(" 11:00"); orari.add(" 11:30"); orari.add(" 12:00"); orari.add(" 12:30");
    orari.add(" 15:00"); orari.add(" 15:30"); orari.add(" 16:00"); orari.add(" 16:30");
    orari.add(" 17:00"); orari.add(" 17:30"); orari.add(" 18:00"); orari.add(" 18:30");


    DateTime start = DateTime.now().minusDays(2);
    DateTime end = start.plusDays(1);
    List<DateTime> between = getDateRange(start, end);


    for (DateTime d : between) {
        for (String o : orari) {
            Bson delete = Updates.pull( fieldName: "reservations", new Document("date", d.toString(DateTimeFormat.shortDate()) + o));

            //Bson delete = delete;
            doctors.updateMany(new Document(), delete);
            users.updateMany(new Document(),delete);

        }}}
```

Figure 13: Delete date function

3.2.5 Indexes

We created 3 MongoDB indexes in order to improve performances of read operations.

The created indexes are:

1. ***Username Doctors*** (single field, field: *username* collection: "*Doctors")*
2. ***Username Users*** (single field, field: *username* collection: "*Users*"
3. ***City and Specialization*** (compound index, fields: *city, specialization* collection: *"Doctors"*)

```
> db.users.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_"
        },
        {
                "v" : 2,
                "unique" : true,
                "key" : {
                        "username" : 1
                },
                "name" : "Username User",
                "background" : false
        }
]
```

Figure 15: Indexes Users

```
> db.doctors.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_"
        },
        {
                "v" : 2,
                "unique" : true,
                "key" : {
                        "username" : 1
                },
                "name" : "Username Doctor",
                "background" : false
        },
        {
                "v" : 2,
                "key" : {
                        "city" : 1,
                        "specialization" : 1
                },
                "name" : "City adn Specialization",
                "background" : false
        }
]
```

Figure 14: Indexes Doctors

The first index is used in different functions, that are:

NAME	DESCRIPTION
add_doctor	Add a new doctor in the DB: Insert a new username at registration time of a Doctor, Check uniqueness of a username at registration time
login_doctor	Check doctors' s credential at login time
delete_doctor_by_the_administrator	Delete a doctor: searching by his username
GetMyProfile	Find a doctor (through his username), print all the information

Table 1: Function Index Username Doctors

The second index was created for the login and sign in of the users. The index is used in different functions, that are:

NAME	DESCRIPTION
add_user	Add a new user in the DBMS: Insert a new username at registration time of a User, Check uniqueness of a username at registration time
login_user	Check user' s credential at login time
delete_user_by_the_administrator	Delete a user: searching by his username

Table 2: Function Index Username Users

The third index was created because we estimate that the queries for retrieving doctors, filtering them by their city and specialization would be frequent. Functions who are involved for this index are:

NAME	DESCRIPTION
cheapestDoc	Find doctors where city and specialization are those requested by the user and print the list of doctors, sorted by price in ascending order

getDocByCitySpec	Find doctors where City and Specialization are those requested by the user and print the list of doctors related.
display_cities	Find all cities of doctors that are registered on the app

Table 3: Function Index City and Specialization

3.2.6 Indexes performances

We analyzed the read-heavy operations on the database and saw which one of them could get some advantage by using certain indexes. We will explain in next paragraphs why we decide to use those indexes, and all performance tests were made in local.

3.2.6.1 *Username Doctors*

A username is a REQUIRED and UNIQUE field of each Doctors, and it is the id inside the application. The username field is used mainly in the login and sign in functions, so the doctor authentication is an operation done with an elevate frequency. We have a few write operations because ones a doctor is stored in our database, we don't need other write operations. When a new doctor sign up, the system check the uniqueness of the username, so assuming we have 100 new doctor at month, we will have 100 read operations. For what concerned the read operations, assuming that a registered doctor need to login in the app almost everyday to see his calendar with the reservations, and at least 2 times a day, and 1000 doctors are using our app every day, we will have 1000*2*30 days of month= 60000 operations for month.

To evaluate if the indexes on Username was necessary, we have used MongoDB's explain clause, and check the performances with and without using the index. We submit this query:

```
db.doctors.find({username:"Giuseppe"}).explain("executionStats")
```

Figure 16: Doctors find

This is the result without index:

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 479,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 2704,
```

Figure 17: performace without index 1

Results with index:

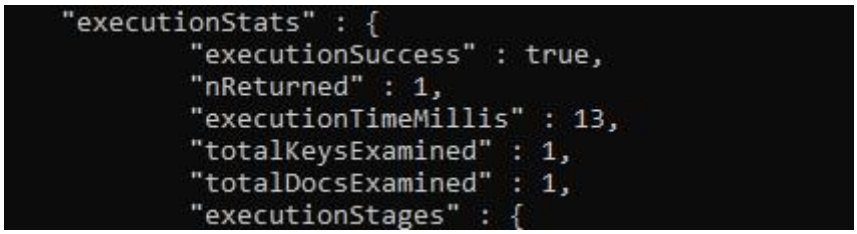

Figure 18: performance with index 1

As we see, the *executionTimeMillis* of the query is 36 times lower with the index, and of course thanks to it, the DBMS needs only to examine one document.

3.2.6.2 *Username Users*

A username is a REQUIRED and UNIQUE field of each Users, and it is the id inside the application.

The username field is used mainly in the login and sign in functions, so the user authentication is an operation done with an elevate frequency. As we explain before with the index on username doctors, when a new user sign up, the system check the uniqueness of the username, so assuming we have 100 new users at month, we will have 100 more read operations in a month. Assuming 250 users are using our app every day, when they login, we will have 250 read operation, so (250*30 days in a month)+ 100= 7600 read operations for month.

As seen before, let us compare DBMS performances with and without a country index. We submit this query:

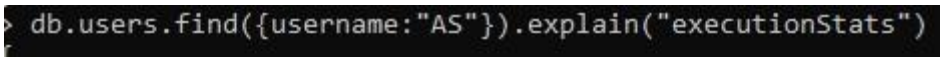

Figure 19: find Users

This is the result without index:

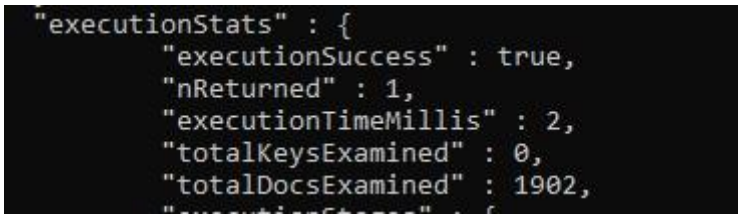

Figure 20: performance without index 2

Results with index:

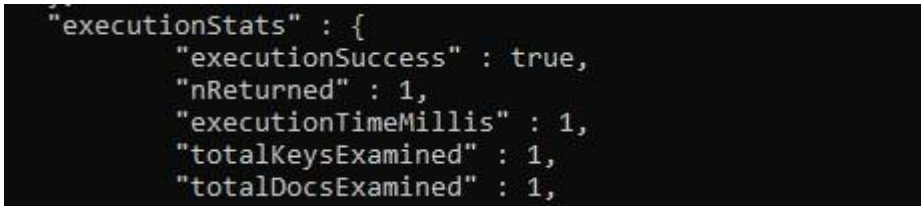

Figure 21: performance with index 2

As we can see, the *executionTimeMillis* is halved, and the DBMS examined only one Docs, so UNIQUE INDEX on username on users has been created.

3.2.6.3 *City and Specialization*

Every doctor has his field *specialization* and a field *city*, those are requested when a doctor sign in, and are not unique. When a user logs in in the app, and chooses to find a doctor, the system will show a list of specialization and city available, then the user will choose a city and a specialization. After that, he will choose what city and specialization wants to see, then he can choose to see the entire list of doctors or the cheapest list. In all these choices, the index on "city" and "specialization is necessary in order to simplify the process. As we said before, these function are used every time a user want to find a doctor, that would be the main reason to use the application. As seen before, let us compare DBMS performances with and without the index. We submit this query:

```
db.doctors.find({city:"Milano"},{specialization:"Dentist"}).explain("executionStats")
```

Figure 22: Find City and Specialization

This is the result without index:

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 694,
        "executionTimeMillis" : 5717,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 2702,
```

Figure 23: performance without index 3

Results with index:

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 694,
        "executionTimeMillis" : 2,
        "totalKeysExamined" : 694,
        "totalDocsExamined" : 694,
```

Figure 24: performance with index 3

In this case there is a considerable boost in performances, passing from 5717 ms of execution time per query without using indexes, to t2ms when using index.

3.3 GRAPH DB

3.3.1 Nodes

The main types of used nodes are:

- **Doctors**: represents the "Doctor" entity. In this type of node are stored the following attributes: doctor's name (*docname*), doctor's username (*doc_id*), city and specialization
- **Review:** represents the review made by a user to a doctor. The stored attributes are: a review_id used to identify a single review, the rating (a number from 1 to 5 that summarises in a quantifiable way the user's overall opinion of the doctor), the text and the date when the review was written.
- **User:** represents the user. In this type of node there is stored only the username to identify the user.

3.3.2 Relationships

The relationships between nodes are:

- **User ---->WRITES---->Review**
- **User---->LIKES---->Review**
- **Review--->BELONGS TO---->Doctor**

All the data present in the Graph DB are real data, except for likes, that was an additional feature added by us, so to populate the DB, we used the application to add some random "like" relationships.

3.3.3 Graph structure

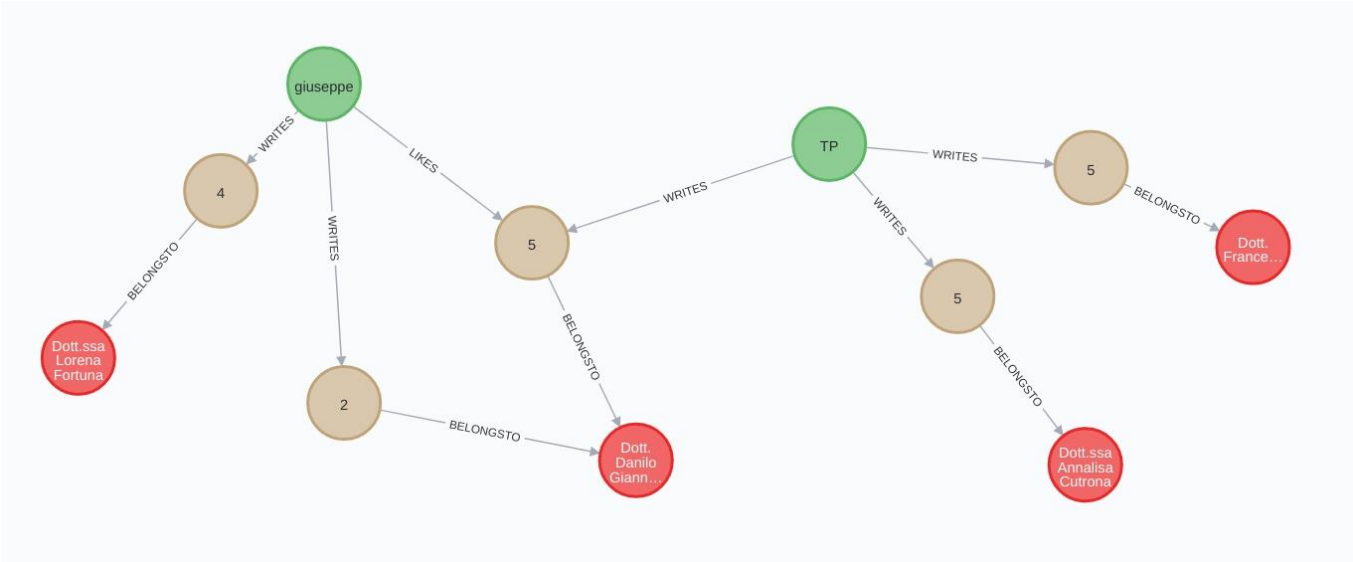

Figure 25: Graph structure

3.3.4 Queries handled

Application Domain	Graph Domain	Expected frequency
Delete a doctor	Delete a DOCTOR node d and all the REVIEW nodes linked to d by the edge BELONGS TO	LOW
Delete a review	Delete a REVIEW node	LOW
Add a review	Add a REVIEW node r, a BELONG TO edge to a DOCTOR node (add it if it doesn't exist) d and a WRITES edge from a USER (add it if it doesn't exist) node to the node r	HIGH
Show all the reviews of a doctor	Match all the REVIEW nodes linked to a given DOCTOR node by an out coming BELONGS TO edge	HIGH
Put a like to a review	Add a relationship (LIKES edge) from a USER node to a REVIEW node	AVERAGE
Ranking of best doctors for each city and specialization	Match all DOCTOR nodes of a given city and specialization. Compute the mean of all "rating" attributes of REVIEW nodes linked to d by a BELONGS TO edge. Compute the total number of REVIEW nodes linked to d. Compute the score for each DOCTOR node. Return the DOCTOR nodes ranked by decreasing "score" value	HIGH
Ranking of "best reviewers"	For each USER node u, compute the number of incoming LIKES edges to	AVERAGE

	REVIEW nodes linked to u by a WRITES edge. Compute the number of out coming "WRITES" edges from u. Compute the ratio between the two computed values. Return all USER nodes ranked by decreasing value of ratio.	
Ranking of most active users	Match all the (USER)-[WRITES]-> (REVIEW) paths, where the difference between the attribute date of REVIEW node and the current date is < 30 (days). Compute the amount of REVIEW nodes linked to the USER node. Return USER nodes ranked by decreasing value of sum of written reviews.	AVERAGE

Table 4: Queries on GraphDB

The implementation of the queries is shown at the "Implementation" paragraph.

3.3.5 Indexes

We expect that, as shown above, that most frequent queries are related to retrieve the reviews of a given doctor, searched by its username, so we decided to create an index on the **doctor's username** to improve query performances.

To demonstrate that the index created is helpful to get better performances, below are showed some statistics running a query, matching a doctor node specifying the username (doc_id).

```
neo4j$ explain MATCH (d:Doctors) WHERE d.doc_id = "5fec4d743a65753d7c4b4ca3" RETURN d
```

Figure 28: Query on index

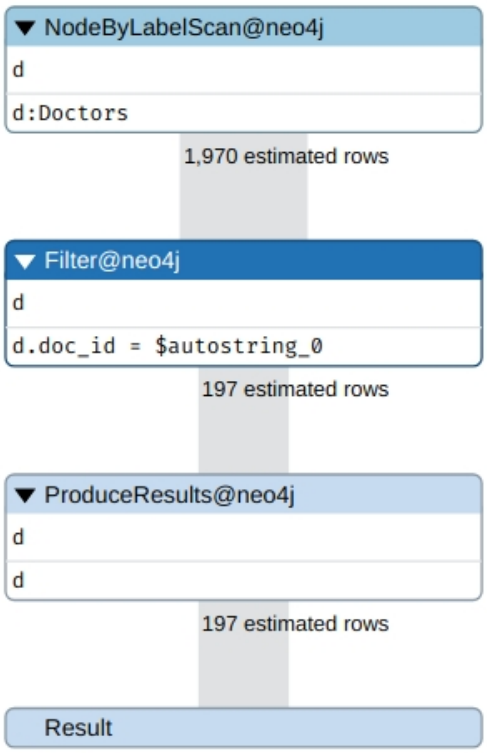

Figure 27: performance without index

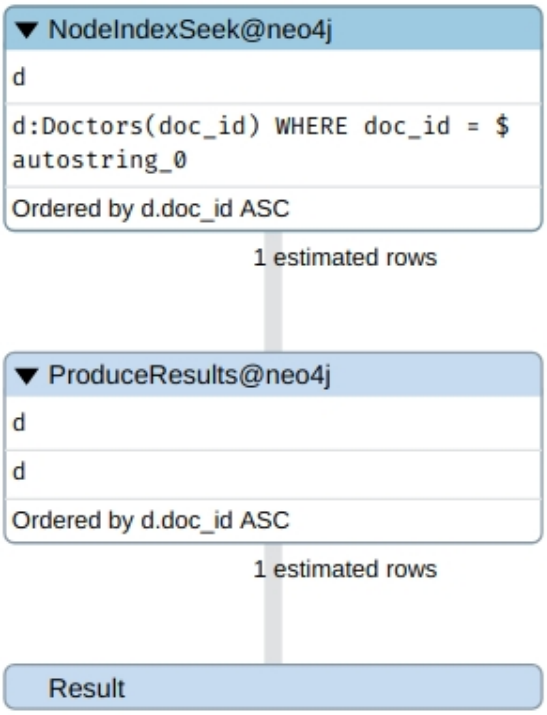

Figure 26: performance with index

We can see a reduction in the number of rows handled.

4 DATABASE PROPRIERTIES

4.1 Redundancies and reservations handling

The main redundancy we have concerns the **reservations.**

As said before, when collections were illustrated, the array of reservation documents is present both in the collections of Doctors and Users.

We avoided to create a single collection for reservations because of queries that have to perform using the application.

Indeed, a single collection of reservations would grow up a lot as the number of new registered doctors increases, considering that a single doctor can add a lot of empty reservations (available slots) for the future, so a doctor, or a user who wants to retrieve and check all his/her reservations (and it's an operation very frequent especially for a doctor that uses regularly the application) would scan a huge number of documents.

For this reason we decided to nest reservations in the document of the relative doctor in order to improve the performance of the query used to retrieve all the doctor's reservations.

Similarly, we decided to add the reservation also inside the user document, so when a user wants to retrieve all his reservations he doesn't have to scan all reservations of all doctors.

The drawback of this choice is having to do a double write operation for each booking or deleting, but we expect that a single user can't suffer from a lot of pathologies, so the number of bookings of medical examinations will be limited if we compare to the number of reads that the same user can do with his own reservations or especially to the number of times that every day a doctor checks the list of his own reservations.

We duplicated also some information about doctors and users to store them also in the Graph Database.

However we store in the graph Database only attributes that are necessary for the correct execution of the queries.

These attributes are: *username* about User and doctor's username, name, city and specialization.

4.2 Consistency, Availability, Partition Tolerance

The **availability** is a non-functional requirement of the application and it is ensured by use of a **cluster of replica set**, hosted in three virtual machines.

The use of the cluster should also guarantee **Partition Tolerance**, so a single node should not cause the entire system to collapse and the cluster should continue to work.

PLEASE NOTE: The replicas for the Graph Database were not deployed on the virtual cluster, but is present a single instance in the virtual machine at the address 172.16.3.110:7687

We cannot ensure a strict consistency, so it could happen that not all the replicas are synchronized with the same data at the same time, but we wanted to ensure **eventual consistency**, especially on data managed by MongoDB, so data involved in the booking process of a medical examination as described in the non-functional requirements of the application.

Indeed, we thought important that a patient who books a medical examination, must see that reservation in his list when he wants to see it. (*read your writes consistency*)

If a user makes several updates on a reservation i.e., a user books a reservation, then free the slot and then books again the same slot, these updates must be done in the order the user issued them. In the previous example if the second operation is performed after the third, the slot would be available for others patients. (*monotonic write consistency)*

When a doctor or a user sees the state of a reservations (if available or not), then all the other users will not see any earlier version of the state of that reservation. (*monotonic read consistency)*

To achieve these kinds of eventual consistency, we set a specific combination of the read and write concerns of the cluster: as described in the MongoDB documentation, write concern and read concern must be set to "*majority"*

Read Concern	Write Concern	Read own writes	Monotonic reads	Monotonic writes
"majority"	"majority"	✔	✔	✔
"majority"	{ w: 1 }		✔	
"local"	{ w: 1 }			
"local"	"majority"			✔

Figure 29: consistency table

From the MongoDB documentation: https://docs.mongodb.com/manual/core/causal-consistency-read-write-concerns/

A booking or a deleting of a reservation, is a write operations that modifies multiple documents in different collections: a reservation booked from a user, as described in the non-functional requirements must be present in the list of reservations of doctors and users, so both in their relative collections.

We achieved the **atomicity** of the whole operation using *Multi-Document Transactions*, supported on replica sets from the version 4.0 of MongoDB: https://docs.mongodb.com/manual/core/write-operations-atomicity/

4.2.1 Cross Database consistency

As we have some duplicated data, for doctors and users, and queries distributed among the two databases we had to think about how to handle the consistency.

In the graph database there is a "Doctor" node, corresponding to the same doctor in a document of the collection Doctors in MongoDB only if he ever received at least one review, and in the same way, a "User" node is present in Neo4j if the user has ever made a review.

Doctor and User nodes are created if they not already exist in the graph database, when the user makes a review for the first time or the doctor receives a review for the first time.

So the only operation we had to manage is the operation of removing a doctor: if a doctor is removed by the administrator, should be removed both from the Document Database and from the Graph Database.

In particular, the problem would be if a doctor exists in the graph Database but not in the document one: the user should not be allowed to see him in the list of best doctors recommended by the system for that given city and specialization, because if it appeared in the list and the user chose that doctor, he could not see his profile, or book a medical examination with him, all operations managed by Mongo DB.

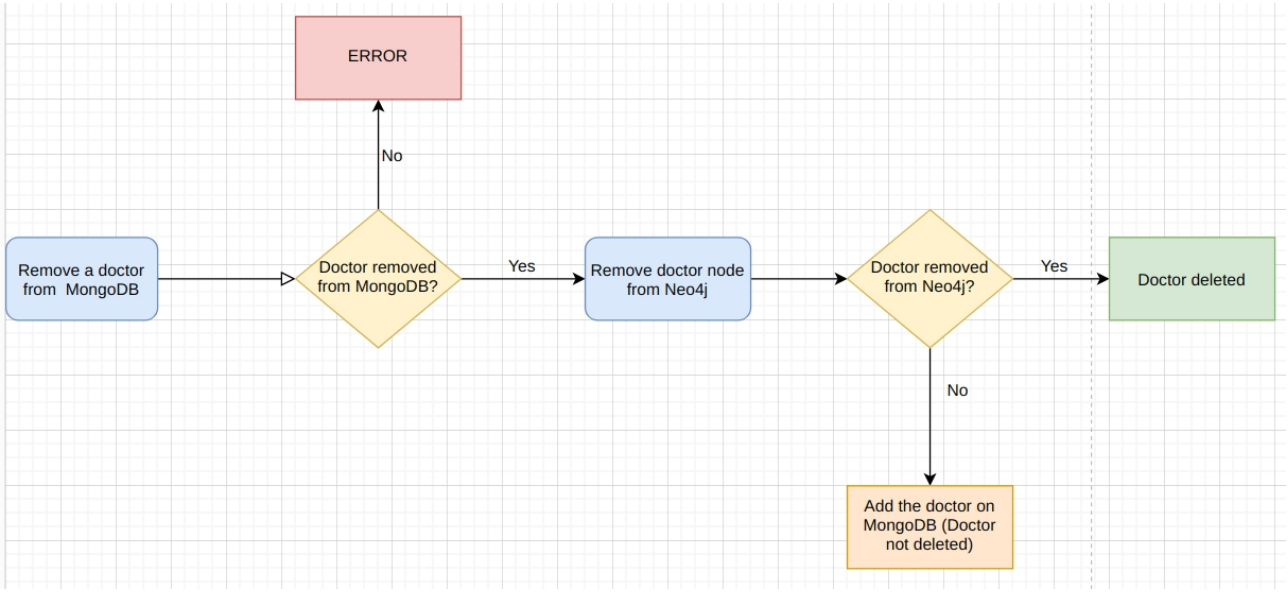

Figure 30: managing cross database consistency

5 IMPLEMENTATION

5.1 Package structure

The Package structure decided for this project is the follow:

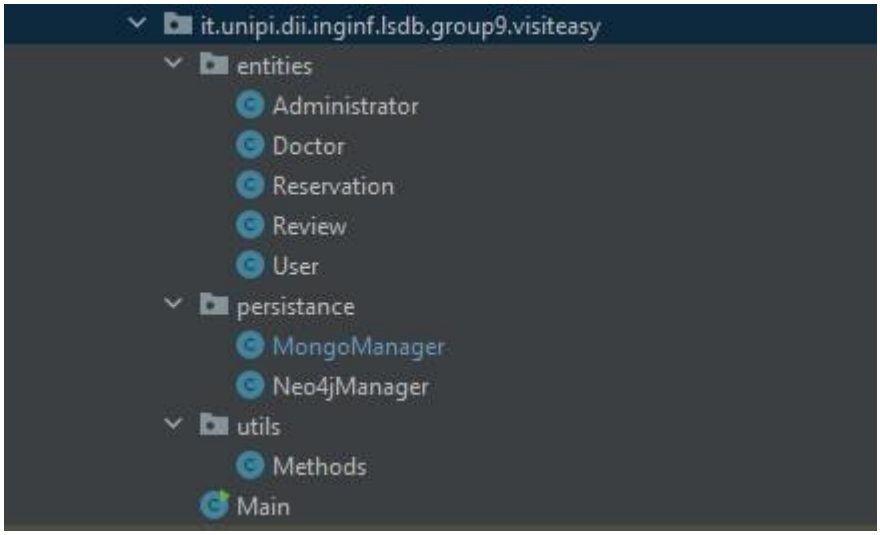

Figure 31: Package

We have not implemented a GUI.

We decided to create three different packages in order to ensure the readability and maintainability of the code. In this chapter will be explained what is inside of each package.

5.1.1 Entities

The package *"entities"* contains all the class related to the actor, used in the code, used in both databases:

- ***Administrator:*** used for instantiating object that refers to a specific administrator
- ***Doctor:*** used for instantiating object that refers to a specific doctor
- ***Reservation:*** used for instantiating object that refers to a specific reservation
- ***Review:*** used for instantiating object that refers to a specific review
- ***User:*** used for instantiating object that refers to a specific user

5.1.2 Persistance

The package "*persistance*" contains two classes:

- **MongoManager**
- **Neo4jManager**

These classes contain all the methods related to the communication with database, respectively MongoDB and Neo4j.

5.1.3 Utils

The package *"utils"* contains the class Methods which contains all the print methods called in the main, in order to keep the main more readable.

5.2 Project Object Model (POM) File

The entire development cycle of the Java application relied on the support of the Maven project management tool, where the Project Object Model (POM) file that was used is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>groupId</groupId>
    <artifactId>VisitEasy</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>8</source>
                    <target>8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>org.mongodb</groupId>
            <artifactId>mongodb-driver-sync</artifactId>
            <version>4.1.1</version>
        </dependency>
        <dependency>
            <version>4.1.1</version>
        </dependency>
        <dependency>
            <groupId>org.json</groupId>
            <artifactId>json</artifactId>
            <version>20090211</version>
        </dependency>

        <dependency>
            <groupId>commons-io</groupId>
            <artifactId>commons-io</artifactId>
            <version>1.4</version>
        </dependency>
        <dependency>
            <groupId>joda-time</groupId>
            <artifactId>joda-time</artifactId>
            <version>2.10.9</version>
        </dependency>
            <dependency>
                <groupId>org.neo4j.driver</groupId>
                <artifactId>neo4j-java-driver</artifactId>
                <version>4.1.1</version>
            </dependency>
    </dependencies>
</project>
```

Figure 32: POM file

5.3 Analytics and Aggregations in MongoDB

In this chapter are listed all the implementations of all the aggregations

Prints all the specializations available in the DB:

```
public void display_spec() //print all specialization available in the DBMS
{
    Bson myGroup = Aggregates.group( id: "$specialization");
    doctors.aggregate(Arrays.asList(myGroup)).forEach(printvalue);
}
```

Figure 33: print specializations available

Prints all the cities available in the DB:

```
public void display_cities() //print all cities present in the DBMS
{
    Bson myGroup = Aggregates.group( id: "$city");
    doctors.aggregate(Arrays.asList(myGroup)).forEach(printvalue);
}
```

Figure 34: print cities available

Analytics: Print the ranking of cities where there are more users, in order to know where the application is used more: (Administrator only)

```
Bson group = group( id: "$city", sum( fieldName: "count",  expression: 1L), Accumulators.push( fieldName: "city",  expression: "$city"));
Bson sort = sort(descending( ...fieldNames: "count"));
Bson limit = limit(3);
Bson project = project(fields(excludeId(), computed( fieldName: "city",  expression: "$_id"), include( ...fieldNames: "count")));


List<Document> results = users.aggregate(Arrays.asList(group, sort, limit, project)).into(new ArrayList<>());
```

Figure 35: Analytics on city

Analytics: Ranking of the three most expensive specializations (Administrator only)

```
//Analytics 3 most expensive specializations
public void printMostExpSpec() {

    Bson group1 = new Document("$group", new Document("_id", new Document("specialization", "$specialization"))
            .append("AvgPrice", new Document("$avg", "$price")));
    Bson project1 = project(fields(excludeId(), computed( fieldName: "specialization",  expression: "$_id.specialization"),
            include( ...fieldNames: "AvgPrice")));


    Bson sort = sort(descending( ...fieldNames: "AvgPrice"));
    Bson limit = limit(3);
    System.out.println("The three most expensive specialization are: ");
    doctors.aggregate(Arrays.asList(group1, project1, sort, limit))
            .forEach(printDocuments);

}
```

Figure 36: Analytics on most expensive specializations

Project only the reservations available of a given doctor, used when the user is going to book a medical examination with a specific doctor:

```
Bson match = match(eq( fieldName: "username",usernamedoc));
Bson unwind = unwind( fieldName: "$reservations");
Bson match2 = match(eq( fieldName: "reservations.patient", value: ""));
Bson project = project(fields(excludeId(), include( ...fieldNames: "reservations")));

doctors.aggregate(Arrays.asList(match,unwind,match2,project)).forEach(printcale);
```

Figure 37: project of Analytics

Analytics: Ranking of the three cheapest doctors filtered by city and specialization, in order to help the user in the choice of a doctor:

```
Bson myMatch = match(and(eq( fieldName: "city",city), eq( fieldName: "specialization", specialization)));
Bson mySort = sort(ascending( ...fieldNames: "price"));
Bson myLimit = limit(3);

doctors.aggregate(Arrays.asList(myMatch,mySort,myLimit)).forEach(addtolist);
```

Figure 38: Analytics cheapest doctors

5.4 Cypher implementation queries in Neo4j

A selection of implementation of queries used in Neo4j in the Cypher language:

Add a review**.** Create nodes of doctor and user if they don't exist:

```
MERGE (u:User{username: $username} )
MERGE (d:Doctors{doc_id: $docname})
CREATE (r:Review{rating: $rating,text: $text, review_id: $id, data: $date})
CREATE (u)-[:WRITES]→(r)-[:BELONGSTO]→(d)
```

Figure 39: Add a review

View all the reviews of a doctor. For each review is computed also the number of likes

```
MATCH (u:User)-[:WRITES]→(r:Review)-[:BELONGSTO]→(d:Doctors)
WHERE d.doc_id = $doctore
WITH r.review_id as id, u.username as username,  r.text as text, r.rating as rating, r.data as date
MATCH (r2:Review)
WHERE r2.review_id = id RETURN id, username, text, rating, date, SIZE(()-[:LIKES]→(r2)) AS NumLikes
```

Figure 40: view all doctor's review

Put a like on a review:

```
MATCH (r:Review) WHERE r.review_id = $review_id
MERGE (u:User{username: $username})
MERGE (u)-[:LIKES]→(r)
```

Figure 41: put like on a review

Analytics: ranking of "best reviewers". (Administrator only)

Best reviewers are users whose reviews are the best: for each user is computed a ratio between the total number of likes received in all his reviews and the total amount of written rewiews.

In this way, a user who has 100 likes with 3 written reviews is cosidered a "better reviewer" then another one who has 120 likes but with 100 written reviews.

```
MATCH (u:User)-[:WRITES]→(r:Review)←[:LIKES]-(u2:User)
WITH u.username AS Username, count(u2) AS NumLikesReceived
MATCH (u3:User)-[:WRITES]→(r3:Review) WHERE u3.username = Username
RETURN u3.username AS Username, toFloat(NumLikesReceived)/count(r3) AS Ratio ORDER BY Ratio DESC LIMIT 3
```

Figure 42: fucntion "better reviewer"

Analytics: Recommended doctors by the system.

When a user is going to search for a doctor, the system will suggest him a list of "best" doctors, filtered by the city and the specialization chosen by the user.

For each doctor belonging to a specific combination of city and specialization, is computed a **score**, based on the average **rating** of all the reviews received and on the total amount of them.

This is the formula used to compute the score:

$$score = \frac{5x}{10} + 5\left(1 - e^{-\frac{a}{b}}\right)$$

Where **x** is the average rating of all reviews which belong to a specific doctor, **a** is the total number of reviews received by the doctor and **b** is a number that shows what is the concept of "a lot of" or "few" reviews. We decided to use b = 10.

So, in this way, a doctor who received only one review with rating = 5, has a score = 2.97, while a doctor who received 20 reviews all with rating = 5 has a score = 6.82, so is considered "better" for the system than the other one.

```
MATCH (r:Review)-[:BELONGSTO]→(d:Doctors)
WHERE d.city = $city AND d.specialization= $specialization
WITH d.docname AS docname, d.doc_id AS us, avg(r.rating) AS AverageRating, count(r) AS NumReviews
RETURN docname, us, 5*AverageRating/10 + 5*(1-exp(-NumReviews/10)) AS score ORDER BY score DESC LIMIT 10
```

Figure 43: Analytics recommended doctors by the system

Analytics: Ranking of the most active users (Administrator only)

PLEASE NOTE: This query is implemented only locally, because it requires the installation of the APOC plugin, that we were not able to install on the virtual machines.

Most active users are users who have written more reviews in the last month.

```
MATCH (u:User)-[:WRITES]→(r:Review)
WHERE (apoc.date.parse(toString(datetime()), 'd',"yyyy-MM-dd'T'HH:mm:ss")
- apoc.date.parse(r.data, 'd',"yyyy-MM-dd'T'HH:mm:ss") < 30 )
WITH u.username AS username, count(r) AS NumWrittenReviews
RETURN username, NumWrittenReviews
ORDER BY NumWrittenReviews DESC LIMIT 10
```

Figure 44: Analytics most active users

5.5 Technologies and Frameworks

The most important technologies and frameworks used to implement our project are:

- ***Java 8:*** the main programming language used. It provides drivers for the communication with databases.

- ***Maven:*** the software project management tool, used mainly for add quickly the dependencies needed.
- ***MongoDB:*** the Document DBMS
- ***Neo4j:*** the Graph DBMS
- ***Cypher:*** The graph query language provided by Neo4j
- ***IntelliJ IDEA:*** the IDE we used to develop the application
- ***Git:*** version control system

The complete implementation of the code is available here: https://github.com/giuseppemartino26/VisitEasy

In the Git repository is also present a **User manual** that can be useful to better understand the functioning of the application.