SMBUD

Contact tracing app

Specification, Entity-Relationship model and Cypher code

POLITECNICO
MILANO 1863

Filippo Lazzati (10629918) - Martina Magliani (10682333) - Christian Grasso (10652464) - Sofia Martellozzo (10623060) - Giacomo Lombardo (10674987)

Year: 2021/2022

Contents

1 Problem specification

The purpose of this project is to build an Information System that monitors and manages data about the COVID-19 pandemic for a specific country. The first part of the project involves the design and development of a Contact Tracing system to monitor the viral diffusion. To be effective, a contact tracing system needs to store information about the contacts between citizens, possibly by keeping track of their relationships (e.g. family, work, etc...). This is key to keep the virus diffusion under control and allow authorities to quickly take action where the situation is critical. To address the problem, we decided to store data in a graph database which allows us to easily manage and visualize the connections and to effectively retrieve significant information.

2 Hypothesis

The database is implemented under the following hypothesis:

- individuals can live together;
- individuals who live together are constantly in contact with each other (contact tracing between them is not needed);
- contacts between two individuals can be registered in two ways:

  through a contact tracing app (e.g. Immuni);

  through public places' registers (e.g. restaurants, theaters, etc. . .);
- for each contact, time and place is stored (when possible); this means that usual contacts through the tracing app do not store the place, but only the time and duration of the contact;
- if a person is tested positive, the system traces his/her direct contacts in the last 5 days;
- if a person has a contact with a positive individual, he/she is notified to take a test: if the test results positive, his/her contacts are traced and notified too;
- individuals who tested positive must take a test (booked by the system) 10 days after the first one. If it is negative they can end their isolation (recovered);
- a test can be moved to another day;
- people in isolation cannot go to other places but their house (for people with more houses, only 1 house is allowed);
- the vaccine campaign has already started:

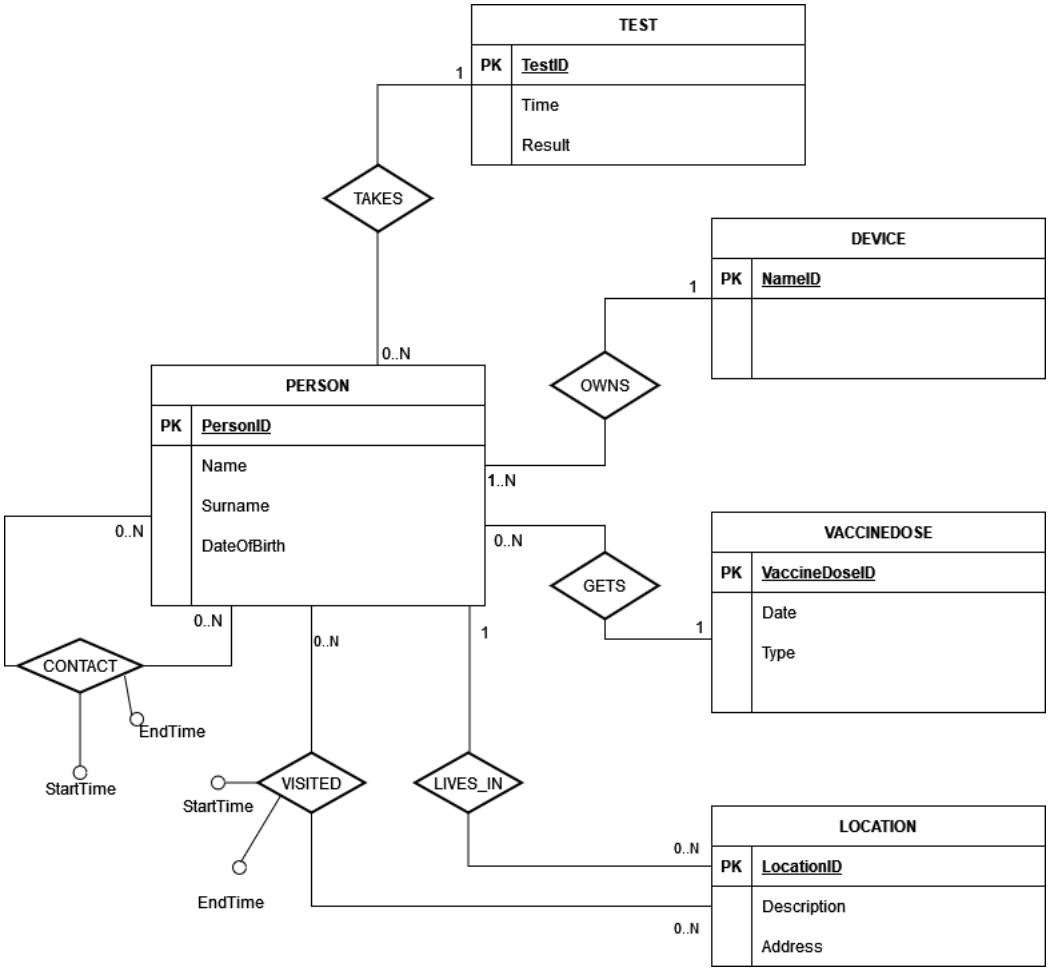
for the non-vaccinated people testing is mandatory;

for the vaccinated people testing is not mandatory (but it is recommended);

- one dose of vaccine suffices to obtain the green pass;
- a test is inserted in the system when booked, but its result is `UNKNOWN` until the test is made.
- a person can own 1 or more devices;
- every device can produce a $CONTACT$ relationship with another person.

3 Conceptual model

The following is the Entity-Relationship (E-R) model of our database:



The Entity-Relationship model contains 5 different entities, that are related to each other through various relationships.

- *PERSON* represents the most important concept of the application, namely the user. Every *PERSON* has a name, a surname and the date of birth. Of course the attributes of such entity could be enlarged, but as a sample dataset we have believed these three are enough;
- *TEST* represents a test a person has to undergo when has been in contact

with a positive (see the assumptions); a *TEST* has 3 possible results, 'positive', 'negative' and 'unknown', where 'unknown' means that the test has been booked but the result is not known yet;

- *DEVICE* represents a device owns by a person on which our contact tracing app is running; for the sake of simplicity it has only one attribute, the *Name*, in addition to the Id;
- *VACCINEDOSE* represents one single dose of vaccine received by a person; it has a date saying when it was administered and the type (brand) of the vaccine;
- *LOCATION* represents either a place where a contact may occur or the home of a person. The attribute *Address* specifies where it is located and *Description* says its type (Home, Museum, ...).

Among these entities, some relations hold. In particular:

- *CONTACT* occurs between 2 different persons; it is recorded by the contact tracing app which specifies also the starting and the ending time;
- *GETS* binds a person with its doses of vaccine;
- *LIVES IN* specifies which person lives in which house. It is used to find people who live together;
- *OWNS* simply relates a person to all his devices. It is a one-to-many relation;
- *TAKES* binds a person with its tests;
- *VISITED* is used to store data about when a certain person was in a certain place (ex.: museum, ...) and at which time, in order to find other people in the same place at the same time.

4 Sample dataset

A sample dataset has been provided along with a file to initialize it. Such sample dataset has been built using some data found through the Google dataset research website at this link.This sample data has been modified and customised using some PHP scripts.
In order to import it into your NEO4J database, you have to put the files contained in the import folder of the project into the *import* folder of the database in Neo4J. Then you can copy and paste the content of the inizialize sample dataset.cql file into the Neo4J browser.In the following there is an explanation of such file.

Since the sample dataset is given in CSV files, you can exploit the LOAD CSV command to load the data (remember to put the files in the IMPORT folder) :

1. Clear the database:

```
MATCH (x) DETACH DELETE x;
```

2. Load the locations:

```
LOAD CSV WITH HEADERS FROM 'file:///locations.csv' AS row
MERGE (l:Location {Id: row.ID, Description: row.Description,
    Address: row.Address});
```

3. Load the persons:

```
LOAD CSV WITH HEADERS FROM 'file:///users.csv' AS row
             MERGE (p:Person {Id: row.ID, Name: row.Name,
    Surname: row.Surname, DateOfBirth: date(row.DateOfBirth),
    HomeLocationId: row.HomeLocationId});
```

4. Create the LIVES IN relationship between users and locations:

```
MATCH
  (p:Person),
  (l:Location)
WHERE p.HomeLocationId=l.Id
CREATE (p)-[r:LIVES_IN]->(l)
```

5. Create the VISITED relationship between persons and locations:

```
LOAD CSV WITH HEADERS FROM 'file:///userlocations.csv' AS row
MATCH (p: Person), (l: Location)
WHERE p.Id=row.PersonId AND l.Id=row.LocationId
CREATE (p)-[:VISITED {StartTime: datetime({epochSeconds:
    toInteger(row.StartTime)}), EndTime: datetime({
    epochSeconds: toInteger(row.EndTime)})}]->(l);
```

6. Load the devices:

```
LOAD CSV WITH HEADERS FROM 'file:///devices.csv' AS row
MERGE (d:Device {PersonId: row.PersonId, Name: row.Name})
```

7. Create the `OWNS` relationship between persons and devices:

```
MATCH
  (p:Person),
  (d:Device)
WHERE d.PersonId=p.Id
CREATE (p)-[r:OWNS]->(d)
```

8. Load the tests' dates with the results (remember: a result can be any of `POSITIVE`, `NEGATIVE` or `UNKNOWN`) :

```
LOAD CSV WITH HEADERS FROM 'file:///test.csv' AS row
MERGE (t:Test {PersonId: row.PersonId, Time: datetime({
    epochSeconds: toInteger(row.Time)}), Result: row.Result});
```

9. Create the `TAKES` relationship between persons and tests:

```
MATCH
  (p:Person),
  (t:Test)
WHERE t.PersonId=p.Id
CREATE (p)-[r:TAKES]->(t)
```

10. Load the contacts between people:

```
LOAD CSV WITH HEADERS FROM 'file:///contacts.csv' AS row
MATCH (p: Person), (q: Person)
WHERE p.Id <> q.Id AND p.Id = row.Person1Id AND q.Id = row.
    Person2Id
CREATE (p)-[r:CONTACT {StartTime: datetime({epochSeconds:
    toInteger(row.StartTime)}), EndTime: datetime({
    epochSeconds: toInteger(row.EndTime)})}]->(q);
```

11. Load the vaccines data about people:

```
LOAD CSV WITH HEADERS FROM 'file:///vaccines.csv' AS row
MERGE (v:VaccineDose {PersonId: row.PersonId, Date: datetime({
    epochSeconds: toInteger(row.Date)}), Type: row.Type});
```

12. Create relationship `GETS` between people and vaccine doses:

```
MATCH (v:VaccineDose), (u:Person)
WHERE v.PersonId=u.Id
CREATE (u)-[r:GETS]->(v)
```

13. Remove the attributes added only for creating the sample dataset:

```
MATCH(p:Person)
REMOVE p.HomeLocationId;

MATCH(t:Test)
REMOVE t.PersonId;

MATCH(v:VaccineDose)
REMOVE v.PersonId;

```

```
10 MATCH(d:Device)
11 REMOVE d.PersonId;
12
13 MATCH(p:Person)
14 REMOVE p.Id;
15
16 MATCH(l:Location)
17 REMOVE l.Id;
```

Now you have a working sample dataset on which you can try the queries of the following paragraph.

5 Queries and Commands

The following queries and commands have been developed in order to provide an example of usage of the system for typical usage scenarios.

5.1 Queries

We have identified the following different queries.
It should be noticed that the 'xxx' ID has been used to select the considered person.

1. **find people with one step of connection**.
   This query can be written without the use of the `UNION` construct in this way:

```
MATCH (a: Person)
MATCH (p: Person)
WHERE id(a)<>id(p) AND id(a)='xxx' AND (
    EXISTS {
        MATCH (a)-[c:CONTACT]-(p)
        WHERE c.EndTime > (date()-duration('P5D'))
    }
    OR
    EXISTS { MATCH (a)-[:LIVES_IN]->()<-[:LIVES_IN]-(p) }
    OR
    EXISTS {
        MATCH (a)-[v1:VISITED]->()<-[v2:VISITED]-(p)
        WHERE v1.EndTime > v2.StartTime and v2.EndTime > v1.
    StartTime and v1.EndTime > (date()-duration('P5D'))
    }
)
RETURN p
```

2. **find people with two steps of connection**.
   In order to perform this query, we could easily repeat twice the previous one:

```
MATCH (a: Person)
MATCH (p: Person)
WHERE id(a)<>id(p) AND id(a)='xxx' AND (
    EXISTS {
        MATCH (a)-[c:CONTACT]-(p)
        WHERE c.EndTime > (date()-duration('P5D'))
    }
    OR
    EXISTS { MATCH (a)-[:LIVES_IN]->()<-[:LIVES_IN]-(p) }
    OR
    EXISTS {
        MATCH (a)-[v1:VISITED]->()<-[v2:VISITED]-(p)
        WHERE v1.EndTime > v2.StartTime and v2.EndTime > v1.
    StartTime and v1.EndTime > (date()-duration('P5D'))
    }
)
MATCH (r: Person)
WHERE id(a)<>id(r) AND (
```

```
    EXISTS {
        MATCH (p)-[c:CONTACT]-(r)
        WHERE c.EndTime > (date()-duration('P5D'))
    }
    OR
    EXISTS { MATCH (p)-[:LIVES_IN]->()<-[:LIVES_IN]-(r) }
    OR
    EXISTS {
        MATCH (p)-[v1:VISITED]->()<-[v2:VISITED]-(r)
        WHERE v1.EndTime > v2.StartTime and v2.EndTime > v1.
    StartTime and v1.EndTime > (date()-duration('P5D'))
    }
)
RETURN r
```

However, in this way we are not able to take into consideration the timing constraints (see the assumptions). Therefore, another way (much longer) to perform the query is to first consider all the people who live with the considered person:

```
match
(p:Person)-[:LIVES_IN]->(l:Location)<-[:LIVES_IN]-(q:Person)
where p<>q and id(p)='xxx'
return q
```

next, we can consider all the people who our person 'xxx' met in the last 5 days:

```
match
(p:Person)-[v1:VISITED]->(l:Location)<-[v2:VISITED]-(q:Person)
where
p<>q and id(p)='xxx' and v1.EndTime > v2.StartTime and v2.
    EndTime > v1.StartTime and v1.EndTime > (date()-duration('
    P5D'))
return q
```

then we can consider all the people who our person 'xxx' has been in contact in the last 5 days according to our system:

```
match
(p:Person) -[c1:CONTACT]- (q:Person), (q:Person) -[c2:CONTACT
    ]- (r:Person)
where
 p <> q and p <> r and q <> r and id(p)='xxx' and
c1.EndTime > (date()-duration('P5D'))
and c2.StartTime > C1.StartTime
and c2.EndTime > (date()-duration('P5D'))
return r
```

finally, we take all the possible combinations ($3^2 = 9$) of these three using the `UNION` construct:

```
match
(p:Person)-[v1:VISITED]->(l1:Location)<-[v2:VISITED]-(q:Person
    ),
(q:Person)-[v3:VISITED]->(l2:Location)<-[v4:VISITED]-(r:Person
    )
```

```
where
 p<>q and p<>r and q<>r and id(p)='xxx'
and v1.EndTime > v2.StartTime
 and v2.EndTime > v1.StartTime
 and v1.EndTime > (date()-duration('P5D'))
and v3.EndTime > v4.StartTime
 and v4.EndTime > v3.StartTime
 and v3.EndTime > (date()-duration('P5D'))
 return r
UNION
match
(p:Person)-[v1:VISITED]->(l:Location)<-[v2:VISITED]-(q:Person)
    ,
(q)-[:LIVES_IN]->(:Location)<-[:LIVES_IN]-(r:Person)
where
 p<>q and p<>r and q<>r and id(p)='xxx'
and v1.EndTime > v2.StartTime
 and v2.EndTime > v1.StartTime
 and v1.EndTime > (date()-duration('P5D'))
 return r
UNION
match (p:Person)-[v1:VISITED]->(l:Location)<-[v2:VISITED]-(q:
    Person),
(q)-[c:CONTACT]-(r:Person)
where
 p<>q and p<>r and q<>r and id(p)='xxx'
and v1.EndTime > v2.StartTime
 and v2.EndTime > v1.StartTime
 and v1.EndTime > (date()-duration('P5D'))
and c.EndTime > (date()-duration('P5D'))
and c.StartTime > v1.StartTime
and c.StartTime > v2.StartTime
 return r
UNION
match
(p:Person)-[:LIVES_IN]->(:Location)<-[:LIVES_IN]-(q:Person),
(q:Person)-[v1:VISITED]->(:Location)<-[v2:VISITED]-(r:Person)
 where
 p<>q and p<>r and q<>r and id(p)='xxx'
and v1.EndTime > v2.StartTime
 and v2.EndTime > v1.StartTime
 and v1.EndTime > (date()-duration('P5D'))
return r
UNION
match
(p:Person)-[:LIVES_IN]->(:Location)<-[:LIVES_IN]-(q:Person),
(q)-[c:CONTACT]-(r:Person)
 where
 p<>q and p<>r and q<>r and id(p)='xxx'
and c.EndTime > (date()-duration('P5D'))
return r
UNION
match
(p:Person) -[c:CONTACT]- (q:Person),
(q:Person)-[v1:VISITED]->(:Location)<-[v2:VISITED]-(r:Person)
where
 p <> q and p <> r and q <> r and id(p)='xxx' and
```

```
c.EndTime > (date()-duration('P5D'))
and v1.EndTime > v2.StartTime
 and v2.EndTime > v1.StartTime
 and v1.EndTime > (date()-duration('P5D'))
and v1.StartTime > c.StartTime
and v2.StartTime > c.StartTime
return r
UNION
match
(p:Person) -[c:CONTACT]- (q:Person),
(q)-[:LIVES_IN]->(:Location)<-[:LIVES_IN]-(r:Person)
where
 p <> q and p <> r and q <> r and id(p)='xxx' and
c.EndTime > (date()-duration('P5D'))
return r
```

Clearly, this query is too long, and it is not able to take into account all the possible timing constraints; thus, the previous version shall be the preferred one.

3. **find people with the Green Pass (i.e. vaccinated people or people with a negative test made in the last 2 days)**.
   First, we have to consider all the people who underwent at least one vaccination (see the assumptions):

```
match (p:Person)-[:GETS]->(v:VaccineDose)
return p
```

   next, we consider all the people who made a negative test in the last 2 days:

```
match
(p:Person)-[:TAKES]->(t:Test)
where
t.Time > (date()-duration('P2D'))
and t.Result = 'negative'
return p
```

   finally, we compute the union of the 2:

```
match (p:Person)-[:GETS]->(v:VaccineDose)
return p
UNION
match
(p:Person)-[:TAKES]->(t:Test)
where
t.Time > (date()-duration('P2D'))
and t.Result = 'negative'
return p
```

4. **find people who violated the isolation**.
   For carrying out this query, the only information we can exploit is the data registered by restaurants, museums, ... According to the assumptions, a person must be at home from the day of a positive test until a negative test, 10 days later:

```
match
(p:Person)-[:TAKES]->(t:Test),
(p)-[v:VISITED]->(:Location)
where
t.Result='positive' and
t.Time > (v.StartTime-duration('P10D')) and
t.Time < v.EndTime
return p
```

5. **show the number of doses distributed for each brand of vaccine**.
   We have to group all the vaccines and count them:

```
match(v:VaccineDose)
return v.Type, count(*) as number_of_doses
```

6. **find people who are healed**.
   This is one of the simplest queries: we look for all the people with both a positive and a negative test, with the negative test made after the positive one:

```
match(p:Person)-[:TAKES]->(t1:Test), (p)-[:TAKES]->(t2:Test)
    where t1<>t2 and t1='positive' and t2='negative' and t1.
    Time < t2.Time return p
```

7. **return all the possible types of location**.
   The type of a Location can be directly accessed from a Location. The `DISTINCT` construct should be used to avoid repeated results:

```
match (l:Location)
return distinct l.Description
```

8. **find all the people who were in place 'xxx' in date 2020-3-3**.
   The fields of a datetime, like the *StartTime* of a `VISITED` relationship can be accessed directly:

```
match (l:Location)<-[v:VISITED]-(p:Person)
where id(l)=12 and v.StartTime.year = 2020 and v.StartTime.
    month = 3 and v.StartTime.day = 3
return p
```

5.2 Commands

We have identified the following `INSERT` commands to show how the system works:

- **insert a new person in the system**.
  I simply use the `CREATE` construct to create such an instance; the only care I have to pay is for the use of the *date*() function:

```
CREATE (p:Person {Name:"nnn", Surname:"sss", DateOfBirth: date
    ("year-month-day")})
```

- **book a test for a person**.
  As it is specified in the assumptions, to book a test in our system a new *Test* node has to be created with 'Unknown' *Result*:

```
match(p:Person)
where id(p)='xxx'
create (t:Test{Result:"Unknown", Time:datetime("2019-06-01T18
    :40:32.142+0100")})<-[:TAKES]-(p)
```

- **insert the result of a test**.
  We use the `SET` construct to change the *Result* attribute of the test (according to the assumptions, no person can have more than one test with 'Unknown' *Result* at the same time):

```
MATCH (t:Test)
WHERE id(t)='xxx'
SET t.Result='Positive/Negative'
```

- **modify the date of a test**.
  We use the `SET` construct to change the *Time* attribute of the test:

```
MATCH (t:Test)
WHERE id(t)='xxx'
SET t.Time=datetime("2019-06-01T18:40:32.142+0100")
```

6 User Interface

6.1 Description

We have provided a demo of a User Interface in the webapp folder in order to show some use cases of the database and how to apply some sample queries/-commands to it. We have implemented a simple client-server application with some basic functionalities to query and update the database. The application is implemented in `JavaScript`, using *node.js* and *express.js* to run the server and the **React** framework for the front-end.

6.2 Functionalities

The application allows to:

- search for a person in the database by his name and surname;
- visualize information for a given person such as name, surname, date of birth, and COVID-19 tests and vaccines;
- insert result of a booked test or edit the test date;
- book a new test;
- show a person's contacts in the last 5 days;

- search for a location by its address or by selecting it from a list of locations of the same type;
- visualize information for a given location and its visitors in a given date;
- visualize basic vaccine distribution statistics.

7 User guide

7.1 Requirements

In order to run the application, the following software need to be installed on your machine.

- Java JDK 11.0
- Neo4j Community Edition
- Node.js

7.2 Database

To run the database:

- Open the Terminal;
- Go to the installation folder of Neo4j Community;
- Go to the bin folder;
- Run ./neo4j console;
- To use the Neo4j User Interface, open your browser and go to http://localhost:7474;
- If you haven't already done it, import the dataset (see section 4).

7.3 Server

After having installed *Node.js*, to run the server

- Clone the github repository;
- Inside the repository, go to graph databases/webapp/webapp-server;
- Edit the config.env file with your Neo4j credentials;
- To install the required node dependencies, use the command:

```
npm install
```

- To run the server, run:

```
npm start
```

7.4 Client

After having started the server, to run the front-end of the application:

- Inside the repository, go to /graph databases/webapp/webapp-client;
- To install the required node dependencies, run:

```
npm install
```

- To run the UI, run:

```
npm start
```

  The browser should open automatically; in case it does not, then go to: `http://localhost:5000`

8 References and sources

- Google Dataset Research
- Kaggle
- draw.io
- node.js
- express.js
- React
- Neo4J
- Java JDK 11

9 Image gallery

In this section we include some screenshots showing the appearance of the User Interface (UI). Read the captions for details about what each image represents.

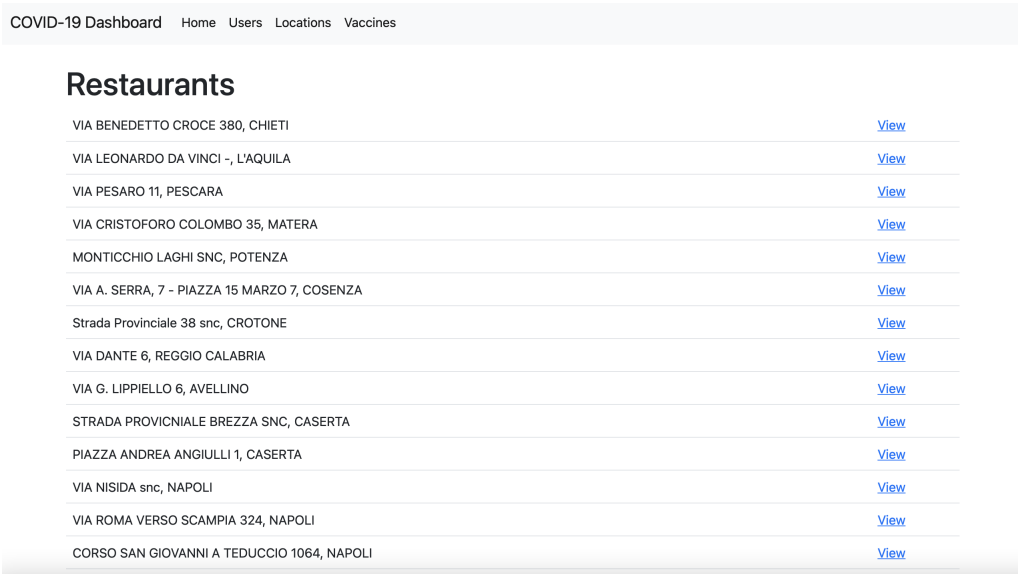


Figure 1: User interface for visualizing all the restaurants in the database.

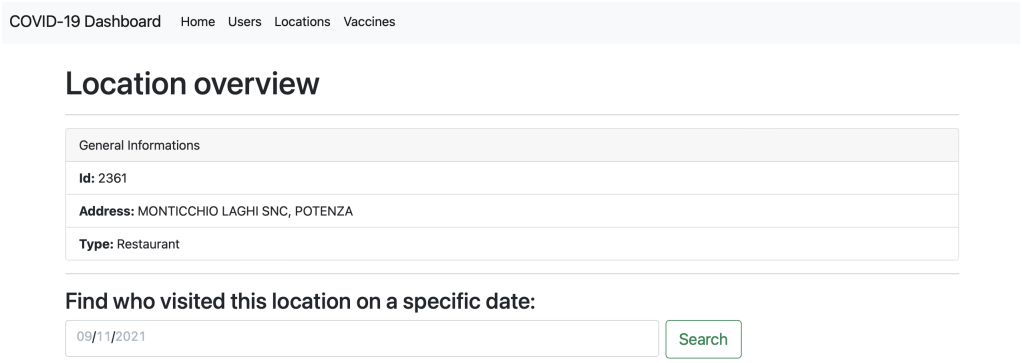


Figure 2: How the app shows the overview of a location.

Figure 3: The contacts of a certain person in the last 5 days.

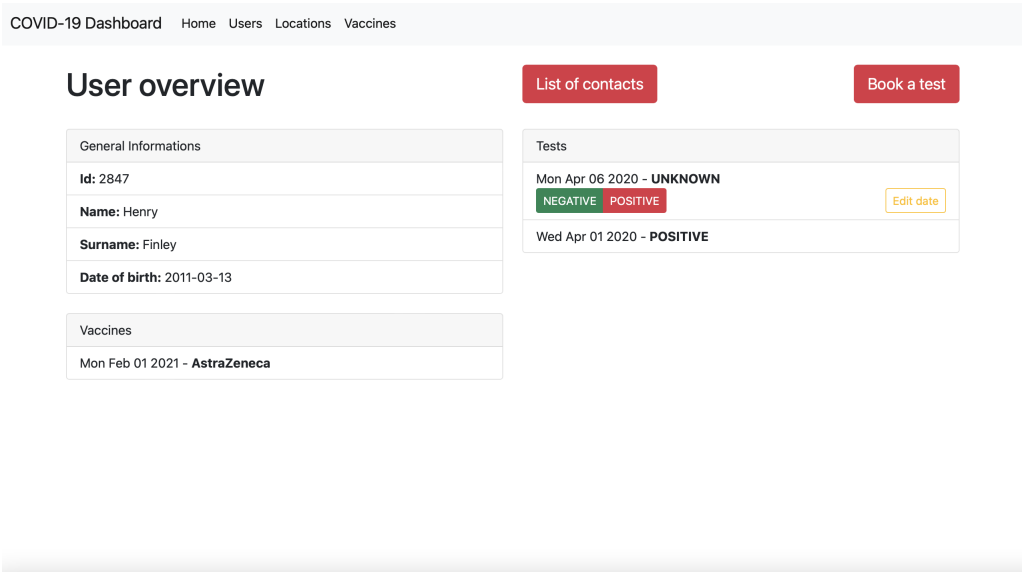


Figure 4: How the app shows the overview of a person.

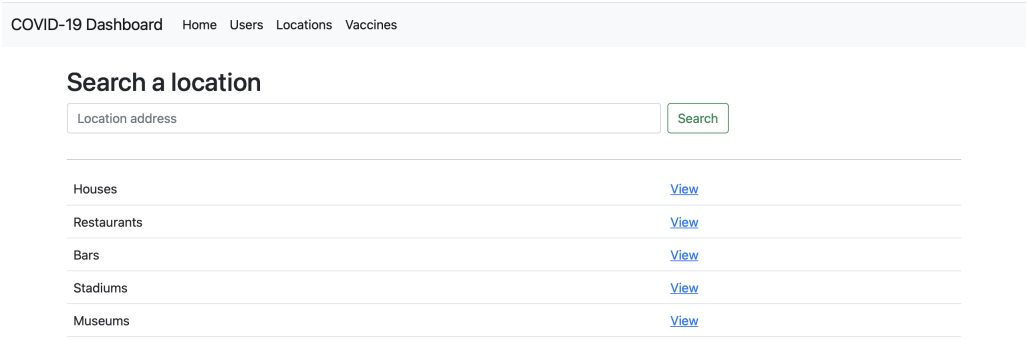


Figure 5: Locations research by address.

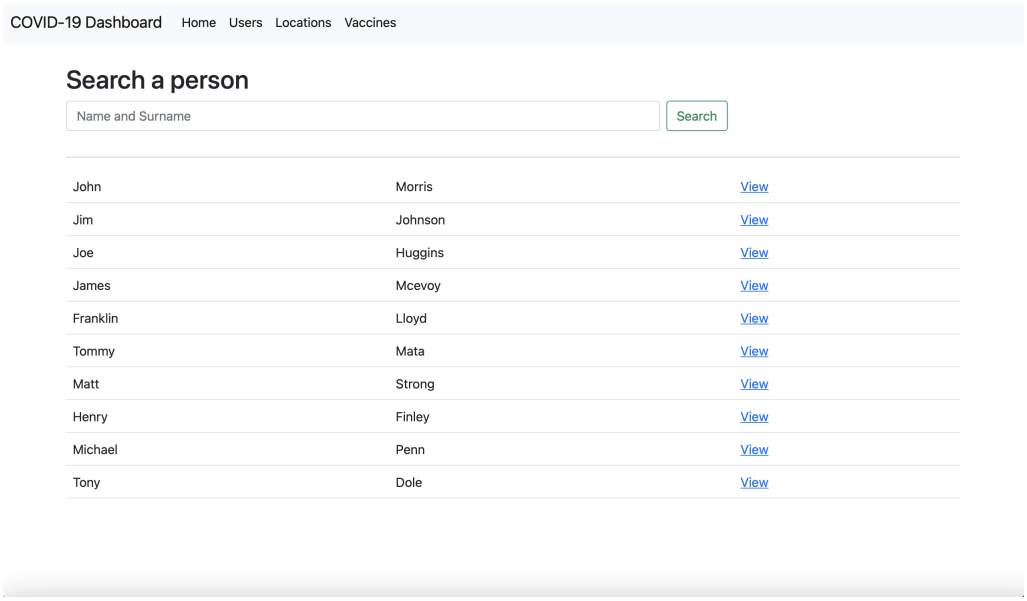


Figure 6: Persons research by name and surname.

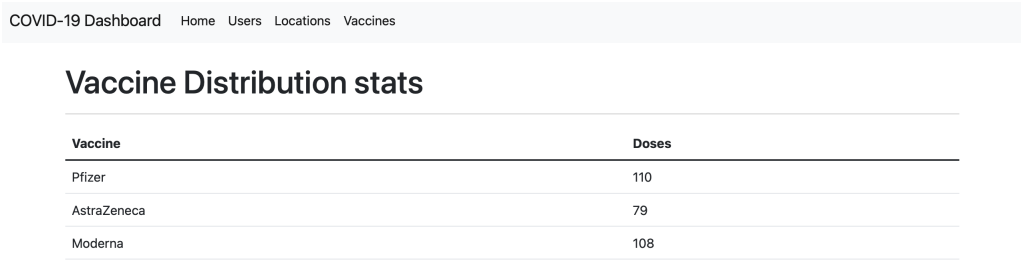


Figure 7: Some statistics about the distribution of vaccine doses.