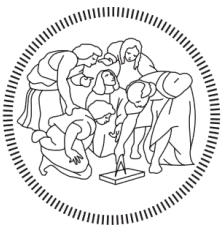


POLITECNICO
MILANO 1863

System and Methods for Big and Unstructured Data

Course project 2021 - Second part

Certificate Management System

Authors

Marcello Bavaro - 10597421

Francesco Puddu - 10583315

Filippo Rescalli - 10609906

Elena Righini - 10796621

Simone Sarti - 10580595

Table of contents

1. Overview

The goal of this project is to build an information system for the management of data related to a pandemic (like the COVID-19 one) in a given country.

In this document we describe the second part of the project, which is the realization of a MongoDB database to support the process of monitoring the validity of health certificates, whose validity will depend on tests performed and vaccines given to each person.

2. Assumptions

- The national health system produces (automatically) exactly one certificate for each person, who is able to access it.

3. Dynamics of the System

When the certificate system is activated for the first time, the national health service will produce, for each citizen, a certificate that will present itself in the form of a QR-code. Every person will be able to show his/her own QR through a device like a smartphone when asked to (events, public transportation, and so on).
When a certificate is shown to the qr-code recognition system, that system will connect to the database and retrieve the document describing the certificate of the person involved. Then, it will show the validity state of that certificate, together with some anagraphic information about its possessor.

The documents that represent the certificates contain in a single place all the useful information to identify the owners of the certificates, plus a log of the tests and vaccines they were subjected to. This log is used to determine the validity of the certificate itself at runtime. In particular, a certificate will be considered valid only if at least one of the following conditions apply:

- The last vaccination (in terms of date performed) has not expired yet.
- The last test (in terms of date performed) is negative and not expired yet .

To make the management system more flexible, a different concept of validity is used for tests and vaccinations. This additional information, that is completely unrelated to the expiration date, is useful to cover some corner cases. In practice, we implemented it by adding to both tests and vaccines a status, which can be:

- Valid: regularly performed
- Invalid: irregularities were discovered, therefore the test/vaccination cannot be considered valid

The two conditions described in the previous paragraph will obviously take into consideration only valid entries from the log.

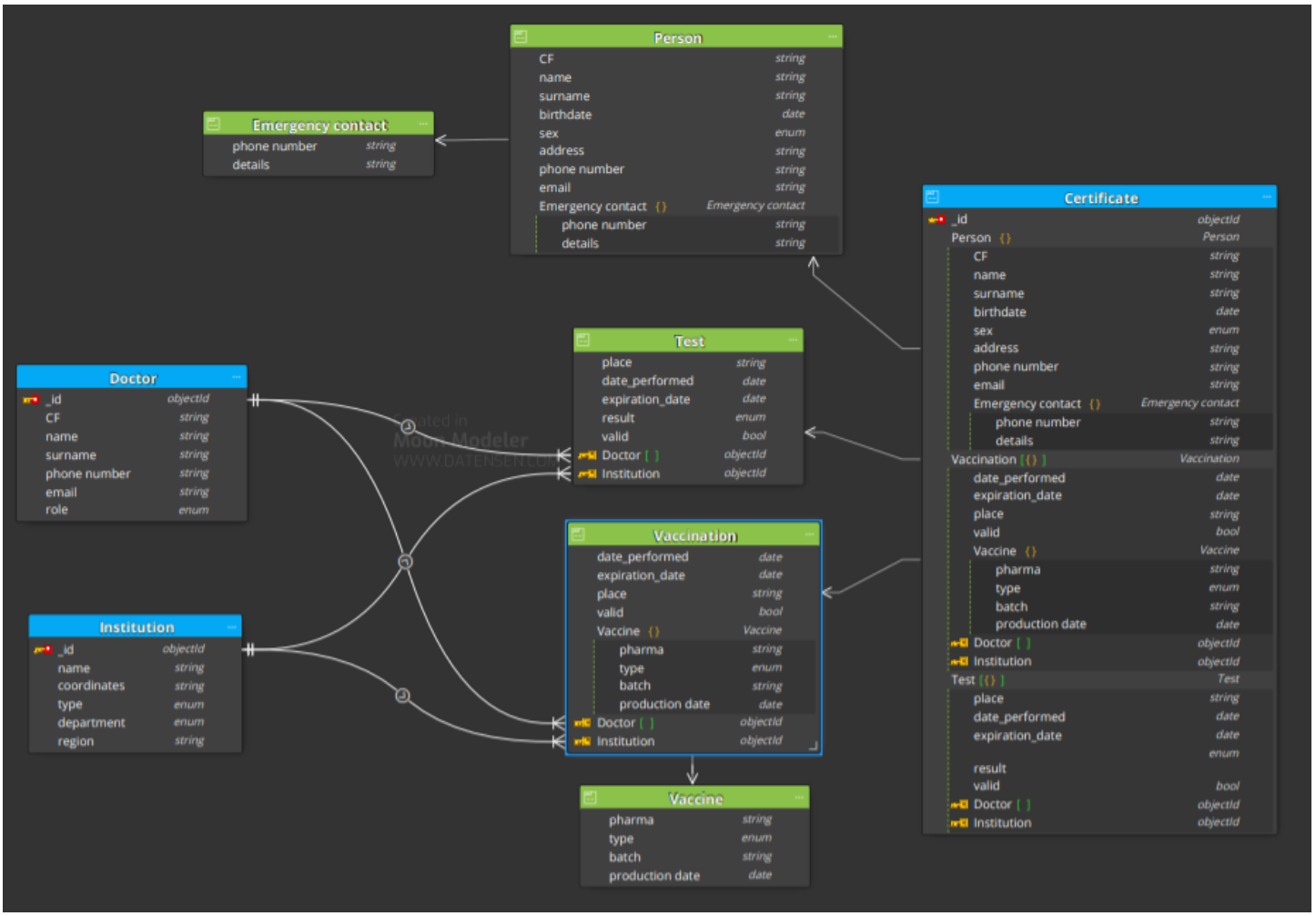
With regard to the expiration of a test or vaccine, while the first will last for a fixed period of 2 days, the duration of the latter will range from 1 to 6 months depending on the number of doses given to the subject (1 month for the first dose, 6 for the followings).

4. Conceptual level diagram

The schema below represents the conceptual level diagram of our system. As shown in the picture, there are 3 independent collections: *Certificate*, *Doctor* and *Institution*.

The *certificate* document contains a subdocument *person* for the anagraphic, a list of subdocuments with the details of the *tests* performed and a list of subdocuments with the details of the *vaccinations* received (these also contain a document describing the *vaccine* used). Moreover, each test or vaccination contains a reference to the relative medical institution and doctors involved.

We decided to include the information of the person inside the Certification document instead of vice-versa because the validation process starts from the scanning of the QR connected to the certificate.



5. Example dataset

We developed a python module to generate a dataset to exemplify our data structure. The user can change different parameters and set them at his/her choice, namely the number of people (or certificates), doctors and institutions to be generated, but also the probability that a test is positive/negative. The basic data (people, doctors and institutions) has been generated by the third-party open tool Mockaroo and then processed mainly using the python library pymongo. This library allows us to connect directly our python script with the mongo server (hosted locally or remotely) and to perform natively mongo's operations like insert, drop and find.

To test our system, we generated a database with a number of nodes in the order of hundreds, setting the parameters in this way:

```
n_people = 500
n_doctors = 20
n_institutions = 40
```

Additional notes on the generated data

- We set the probability of testing positive to 10%. While this is quite unrealistic, it's also useful to avoid having too few positive tests.
- The generator models a situation in which the second dose of a vaccine is always taken 30 days after the first, while from the third onwards the interval ranges between 170 and 200 days.
- To avoid the risk of tests being irrelevant in the queries because of their short validity period, the generator creates many more of them with respect to vaccinations. That's the reason why the first query will show such a large percentage of certificates are valid only because of tests.

The code is available on github.

6. Queries and commands

This section covers our analysis of the interaction with the database. According to the specification, we prepared 5 queries and 3 commands. Given the complexity of the queries, we decided to design them as a combination of many subqueries, that are detailed in the following section. We'll refer to the high-level queries as Qx (Q1, Q2 ...) and to the relative subqueries as Qx.y (Q1.1, Q1.2 ...). The colors are used to help the reader in the mapping between queries and subqueries in section 6.2

6.1 Subqueries

Q1.1 - Q2.1

Returns an array of documents, each containing the id of a certificate of a person for whom the last test exists, is valid, is negative and is not expired.

```
db.Certificate.aggregate([
      {$project: {
        TEST:
            {$filter:{
               input: "$TEST",
               as: "test_i",
               cond:{$eq:["$$test_i.valid", true]}
      }}}},
      { $project: {TEST: { $slice: [ "$TEST", -1 ] } } },
      {$match : {"TEST":{$size:1}}},
      {$unwind: "$TEST"},
      {$addFields:{today:new Date()}},
      {$match : {"TEST.result":"negative"}},
      {$match: {$expr:{$gte:["$TEST.expiration_date","$today"]}}},
      {$project:{"_id":1}}
]).toArray()
```

First of all we filter out from the array of tests of each person the tests that are invalid (using $filter). Then, using $slice:-1, we only keep the latest test. Tests are inserted in the DB in order of date_performed, so we don't need to order them in the query. After these first two operations people without a valid test will have an empty TEST array, we only keep those documents for which the resulting TEST array still contains one element using $match and $size:1. Then, we only keep certificates for which the last remaining test is negative and not expired yet (this requires using $addFields to create a field containing today's date). The $unwind is necessary because for some reason $gte doesn't allow to compare fields at two different levels in the document ("`TEST.0.expiration_date" vs "$today`" simply doesn't work), so the TEST array must be unrolled to allow the comparison. $project makes sure that we return only the id of the documents that survived up to this point, and therefore satisfy the condition.

Q1.2

Returns an array of documents, each containing the id of a certificate of a person that doesn't have a valid vaccination (or has none at all).

```
db.Certificate.aggregate([
      {$project: {
         VACCINATION:
            {$filter:{
               input: "$VACCINATION",
               as: "vaccination_i",
               cond:{$eq:["$$vaccination_i.valid", true]}
   }}}},
   {$project: {VACCINATION: { $slice: [ "$VACCINATION", -1 ] } } },
   {$match : {"VACCINATION":{$size:0}}},
      {$project:{"_id":1}}
]).toArray()
```

To start, we filter out from the array of vaccinations of each person the vaccines that are invalid (using $filter). Then, using $slice:-1, we only keep the latest vaccination. Vaccinations are inserted in the DB in order of date_performed, so we don't need to order them in the query. After these first two operations people without a valid vaccination will have an empty VACCINATION array, and we keep exactly those documents using $match and $size:0. We use $project to return just the id of those documents.

Q1.3

Returns an array of documents, each containing the id of a certificate of a person for whom the last vaccination exists, is valid, but has expired.

```
db.Certificate.aggregate([
   {$project: {
         VACCINATION:
            {$filter:{
               input: "$VACCINATION",
               as: "vaccination_i",
               cond:{$eq:["$$vaccination_i.valid", true]}
   }}}},
   {$project: {VACCINATION: { $slice: [ "$VACCINATION", -1 ] } } },
   {$match : {"VACCINATION":{$size:1}}},
   {$unwind: "$VACCINATION"},
   {$addFields:{today:new Date()}},
   {$match: {$expr:{$lt:["$VACCINATION.expiration_date","$today"]}}},
   {$project:{"_id":1}}
]).toArray()
```

To start, we filter out from the array of vaccinations of each person the vaccines that are invalid (using $filter). Then, using $slice:-1, we only keep the latest vaccination. Vaccinations are inserted in the DB in order of date_performed, so we don't need to order them in the query. After these first two operations people without a valid vaccination will have an empty VACCINATION array, we only keep those documents for which the resulting VALID array still contains one element using $match and $size:1. Then, we only keep certificates for which the last remaining vaccination has expired (use $addFields to create a field containing today's date). The $unwind is necessary for the reason explained in Q1.1 . $project makes sure that we return only the id of the filtered documents.

Q1.4 - Q2.2

Returns an array of documents, each containing the id of a certificate of a person for whom the last vaccination exists, is valid, and is not expired.

```
db.Certificate.aggregate([
      {$project: {
        VACCINATION:
            {$filter:{
               input: "$VACCINATION",
               as: "vaccination_i",
               cond:{$eq:["$$vaccination_i.valid", true]}
    }}}},
      {$project: {VACCINATION: { $slice: [ "$VACCINATION", -1 ] } } },
      {$match : {"VACCINATION":{$size:1}}},
      {$unwind: "$VACCINATION"},
      {$addFields:{today:new Date()}},
      {$match: {$expr:{$gte:["$VACCINATION.expiration_date","$today"]}}},
      {$project:{"_id":1}}
]).toArray()
```

It works just like Q1.3, but in this case the vaccination must not have expired yet.

Q1.5 - Q2.3

Returns an array of documents, each containing the id of a certificate, for all the certificates in the DB. It is used to compute the complement of Q2.4.

```
db.Certificate.aggregate([
      {$project:{"_id":1}}
]).toArray()
```

It simply uses $project to eliminate all unnecessary information present in the documents apart from their ids.

Q1.6 - Q2.4

Returns an array of documents, each containing the id of a certificate of a person for whom the last test exists, is valid and is positive. That means that the person is currently infected.

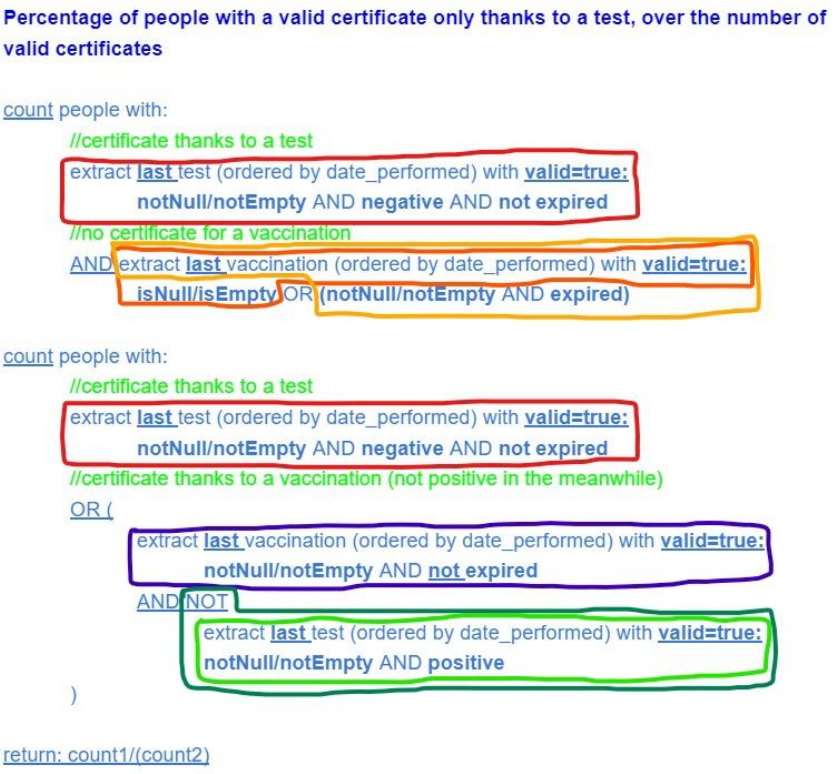
```
db.Certificate.aggregate([
   {$project: {
        TEST:
            {$filter:{
               input: "$TEST",
               as: "test_i",
               cond:{$eq:["$$test_i.valid", true]}
    }}}},
   { $project: {TEST: { $slice: [ "$TEST", -1 ] } } },
   {$match : {"TEST":{$size:1}}},
   {$match : {"TEST.0.result":"positive"}},
   {$project:{"_id":1}}
]).toArray()
```

It works in the same way as Q1.1, but here we consider positive tests instead of negative ones, and there is no need to check expiration dates because a person is considered infected until a successive valid negative test is added to the certificate.

A note on the results
The results of these queries are just arrays of documents containing ids, thus including screenshots of the outputs would not have been meaningful.

6.2 Queries

Query 1 (Q1.1, Q1.2, Q1.3, Q1.4, Q1.5, Q1.6)



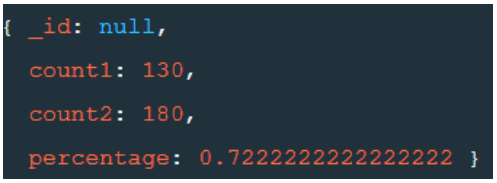

```
db.Certificate.aggregate([
      {$group:{_id:null}},
      {$addFields:{
         part1_array: {$setIntersection:[Q1.1,{$setUnion:[Q1.2,Q1.3]}]},
         part2_array:
             {$setUnion:[Q1.1,{$setIntersection:[Q1.4,{$setDifference:[Q1.5,Q1.6]}]}]},
      }},
      {$addFields:{
         count1:{$size:{"$ifNull": [ "$part1_array", [] ]}},
         count2:{$size:{"$ifNull": [ "$part2_array", [1]]}}
      }},
      {$project:{count1:1, count2:1, percentage: { $divide: [ "$count1", "$count2" ]}}}
])
```

The query is divided into 2 main parts:

- Part 1: computes the number of people who have a valid certificate only due to tests, which means that they must not have a vaccination that would grant them the certificate as well;
- Part 2: computes the number of people who have a valid certificate, regardless of the fact that they have it through tests or vaccinations. Note that a person that is currently positive is not counted here, regardless of the fact that he/she has a valid vaccination, because, as specified in the introduction, positive tests are more "powerful" than vaccinations and they make the certificate invalid until a successive test shows that the person has recovered.

The precise logical conditions that must be met by a certificate to be part of one of the two counts (part1 or part2) are explicitly shown in the image above, together with the subqueries that check each of them.
To obtain the counts, the solution we came up with is to use multiple subqueries to check single conditions, each subquery returns an array composed of a set of documents, each containing the _id of a certificate that satisfies the condition. Given these arrays, we took advantage of the set operators provided by mongo to perform the logical operations: $setIntersection to perform ANDs, $setUnion for ORs, and $setDifference for the NOT.

To save the results of these operations, we had to employ the $addFields command, which allows us to define new fields in a document, and to those fields (which we later used as variables), we assigned the resulting arrays. Since we didn't care about single documents in the computation of the percentage, the first operation of the aggregation is a group by null, which just gives as result an empty document with null id. We learned the hard way that $addFields doesn't create the field until the section is completed, so the assignments to the counts taking the size of the arrays kept failing. When we realized that, we just moved their declaration in a succcessive $addFields. The $ifNull is used to catch exceptions. Once the counts are ready, we return them together with the percentage, computed using the $divide operator.



Query 2 (Q2.1, Q2.2, Q2.3, Q2.4) - Percentage of currently valid certificates in the DB

It returns the percentage of currently valid certificates in the database, by counting how many certificates are currently valid over the total number of certificates in the DB. To count we can take advantage of some of the subqueries we wrote and explained previously. This query allows us to compute another important metric that can show the effectiveness of the measures taken to contrast the disease.

```
db.Certificate.aggregate([
      {$group:{_id:null}},
      {$addFields:{
         part1_array:
            {$setUnion:[Q2.1,{$setIntersection:[Q2.2,{$setDifference:[Q2.3,Q2.4]}]}]},
         part2_array: Q2.3,
      }},
      {$addFields:{
         count1:{$size:{"$ifNull": [ "$part1_array", []]}},
         count2:{$size:{"$ifNull": [ "$part2_array", [1] ]}}
      }},
      {$project:{count1:1, count2:1, percentage: { $divide: [ "$count1", "$count2" ]}}}
])
```

In this query we do not care about specific certificates and their information, so we simply group by a null id to remove them and obtain a blank document that we can use to calculate the result we want.
To obtain the two counts needed to compute the percentage, we define two fields thanks to the $addFields command, and we use them as variables. To compute how many certificates are currently valid, we can use the set operations as we did in the second part of Query 1, instead the computation of the second count can be done exploiting Q2.3 on its own, which came very in handy as we can use this subquery in both part of the query, where it covers different roles. Once again we couldn't directly calculate the counts as sizes of the returned arrays in the same $addFields which performed the subqueries, so we had to create a second addFields. To conclude the two counts and the result percentage are returned.

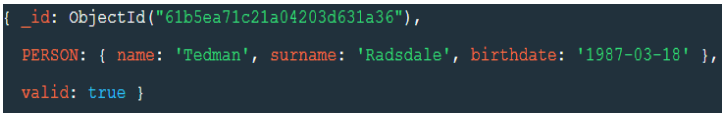
```
{ _id: null, count1: 180, count2: 500, percentage: 0.36 }
```

Query 3 (Q1.1*, Q1.4*, Q1.6*) - Certificate validity check

Given as input the ID of a certificate, returns a document containing name/surname/birthdate of the person linked to that certificate, plus the validity state of the certificate itself. Q1.1*, Q1.4*, Q1.6* are alterations of the original subqueries to consider only one particular certificate.

```
db.Certificate.aggregate([
	{$match:{"_id":ObjectId("XXXX")}},
	{$addFields:{
			array_Qa: Q1.1*,
			array_Qb: Q1.4*,
			array_Qc: Q1.6*,
	}},
	{$addFields:{
			countA:{$size:{"$ifNull": [ "$array_Qa", [1,2]]}},
			countB:{$size:{"$ifNull": [ "$array_Qb", [1,2]]}},
			countC:{$size:{"$ifNull": [ "$array_Qc", [1,2]]}},
	}},
	{$addFields:{
			valid:{$cond:
					{if:
						{$or:[
							{$eq:["$countA",1]},
							{$and:[
								{$eq:["$countB",1]},
								{$eq:["$countC",0]}
							]}
						]},
						then:true,else:false
					}
				}
	}},
	{$project:{
			"PERSON.name":1,
			"PERSON.surname":1,
			"PERSON.birthdate":1,
			"valid":1
	}}
])
```

We only consider one specific certificate this time, so we discard the others. After that, we create three fields acting as variables for our query, they will contain the results of the three subqueries. Because of the fact that the subqueries too consider only that certificate, the returned array will contain 1 element if the certificate meets the query condition, 0 if it doesn't, the idea is to exploit these values as boolean conditions for checking the validity of the certificate itself. The validity conditions are the same as before, but its computation is done in a totally different way, exploiting the $cond and $eq operator. Here too assignments must be split from the counts and from the condition check because of the behavior of $addFields. Only the relevant personal data of the person and the certificate validity status are returned using $project.



Query 4 - AntiVaxxer workers

It returns the list of ids of people who have never been vaccinated and got tested at least 12 times in the last month (considering that a test gives two days of validity for the certificate and excluding the need to have it during weekends, 12 tests should cover the month). The query also orders according to the number of tests performed in descending order.

```
db.Certificate.aggregate([
  {$match : {"VACCINATION":{$size:0}}},
  {$unwind: "$TEST"},
  {$match: {"TEST.date_performed":{"$gte" : new Date(ISODate().getTime() - 1000 * 3600 * 24
* 30)}}},
  {$group: {_id:"$_id", tests_taken:{$sum:1}}},
  {$match:{tests_taken:{$gte:12}}},
  {$sort: {tests_taken:-1}}
])
```

We exclude all those people for whom the list of vaccinations is not empty (they got vaccinated at least once, we also remove those people who had their vaccination invalidated because at least they tried to get the vaccine). We then unroll the TEST array into multiple documents associated with each certificate, and discard all the tests that date back to more than a month ago. We then re-group by the certificate id to count how many tests are left in the current month. (No instances for >= 12 tests, the result below is for >=6, top 5 shown)

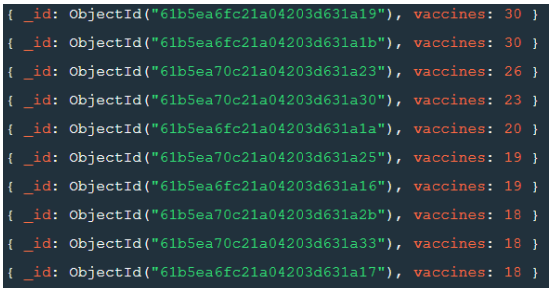
```
{ _id: ObjectId("61b5eac8c21a04203d631b3d"), tests_taken: 10 }
{ _id: ObjectId("61b5eaabc21a04203d631ade"), tests_taken: 10 }
{ _id: ObjectId("61b5ea7fc21a04203d631a5f"), tests_taken: 10 }
{ _id: ObjectId("61b5eaa2c21a04203d631acb"), tests_taken: 8 }
{ _id: ObjectId("61b5eabdc21a04203d631b1e"), tests_taken: 8 }
```

Query 5 - Institutions responsible for the most vaccinations

The query returns the list of ids of the 10 institutions which supervised the largest amount of valid vaccinations, together with the number. Results are ordered in a descending order according to the number of vaccinations.

```
db.Certificate.aggregate([
      {$project: {VACCINATION:1}},
      {$unwind: "$VACCINATION"},
      {$match: {"VACCINATION.valid":true}},
      {$group: {_id:"$VACCINATION.Institution", vaccines:{$sum:1}}},
      {$sort: {vaccines:-1}},
      {$limit:10}
])
```

First we remove from the certificates information that doesn't concern vaccinations. Then we unwind the VACCINATION array and discard all invalid vaccinations. To conclude we group the documents left by the institution that supervised the vaccine, and count how many vaccines are related to each institution, we sort the results, and keep the top 10.



6.3 Commands

Command 1 - Insertion of a test and ordering

The command inserts a new test in a certificate and it makes sure that tests are ordered according to the date in which they were performed.

```
db.Certificate.updateOne(
  {_id: ObjectId("XXXX")},
  {$push:{
       TEST:{ $each:[{
                      "place": "Test Center n.122",
                      "date_performed": new Date("2019-09-17T16:00:00Z"),
                      "expiration_date": new Date("2019-09-19T16:00:00Z"),
                      "result": "positive",
                      "valid": true,
                      Doctor: [ObjectId("XXXXX")],
                      Institution: ObjectId("XXXXX")
                      }],
              $sort:{date_performed:1}
   }}}
)
```

First of all we match the certificate with the id specified in the "_id" field. Then we push in the TEST array a new test specifying all properties, such that place, date_performed, duration,....
Then the array of tests gets sorted.

Command 2 - Invalidation of tests performed by a particular doctor

The command updates some certificates modifying the "valid" field in those tests in which in the doctor equipe is present a doctor that on purpose fakes tests.

```
db.Certificate.updateMany(
      {"TEST": {
            "$elemMatch": {
            "Doctor" : { "$in" : [ObjectId('XX')]}
      }}},
      {"$set":
            {"TEST.$[outer].Doctor.$[inner]": ObjectId('XX'),"TEST.$[outer].valid":false}
      },
      {"arrayFilters":
            [{"outer.Doctor":
                  {"$in":[ObjectId('XXXXX')]}},{"inner":{"$in":[ObjectId('XXXXX')]}
            }]
      }
)
```

First of all we filter for those certificates for which in the TEST arrays is present at least a test that has in the doctor equipe the "mad" doctor. Then we use the "$set" operator to update the "valid" field of the tests to false. In particular using the "arrayFilters" we limit the modification to those tests in the array TEST that have the "mad" doctor _id in the Doctor array (N.B. if we remove the "arrayFilters" the command will update all the tests in the TEST array, even if not all doctor arrays in those tests contain the mad doctor, just because at least one test was performed by the "mad doctor", as specified by the first filter clause).

Command 3 - Update of an emergency contact of a person

The command updates the certificate with the specified "_id" field, changing the emergency contact of that person.

```
db.Certificate.updateOne(
      {"_id": ObjectId("XXXXX")},
      {$set:{"PERSON.EMERGENCY_CONTACT.phone_number":"XXX-XXX-XXXX",
                  "PERSON.EMERGENCY_CONTACT.details":"XXXXXXXXXXXXXXXXXX"}}
)
```

First of all we match the certificate with the id specified in the "_id" field. Then with the "$set" operator we modify the emergency contact of that person.

A note on the results

The results of the commands are not very interesting, as mongo just returns the number of matched and modified documents, but not the result of the operations. Consequently, we decided not to include any screenshot in this report.

7 Optional part

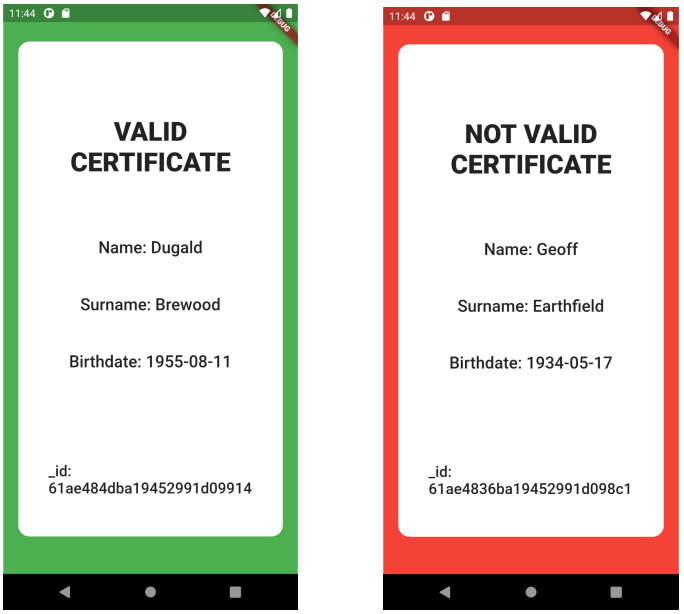
7.1 Mobile application

For the optional part of the project we decided to implement a mobile application using Flutter. The app is meant to simulate the verification of the user certificate validity through the qr code scan mechanism. For what concerns the database interaction we used Atlas since it exposes useful http APIs to manage the data. Moreover, having a shared http reachable database instance allowed us to deploy the application also on mobile devices.
Given that the goal of the app is validating certificates, the query involved is Q3, which gives as result a document containing the information of a person and the boolean flag storing the validity of the associated certificate. We decided to store the output of the query in a separate collection called "ValidCertificates" in order to keep the query made by the application simpler (we previously tried to perform Q3 directly through the application, but the complexity of the query didn't allow us to match the pattern requested by the API call). The collection is not part of the standard functioning of the database, it was just meant to accomplish the mobile app.
In our implementation, the qr codes are actually generated by another tool developed by us and described in the following section.

The application is divided in two main pages:

- **Page 1:** this page is responsible for scanning the qr code and it has been implemented using a dart package that allows to decode the qr codes. Each code is associated with the _id of a particular certificate, once the qr code is recognized a find operation is executed to retrieve the information associated with the certificate.
- **Page 2:** the transition to this page is triggered when the qr code is recognized. It's responsible for displaying the data extracted by the database (like name, surname and birthdate of the certificate owner). The validity of the certificate is denoted by the background color of this page, that will be red for invalid certificates and green for the valid ones.

Here we have some snapshots of the app running on an Android Emulator, however the app was also tested using the apk file running on Android mobile devices. The project folder and the apk file can be found in our Github repository here. Please note that the core of the application is contained in the main.dart file located in the lib folder.



7.2 QR code generator

As mentioned at the beginning of this document, we designed the system relying on the assumption that each person will be given a QR code to reference the personal health certificate. As a companion to the app, we consequently decided to develop a tool to generate such code, starting from the anagraphic data of a person.

The component, namely the script qr_generator.py (available here), can be used as standalone or as a library, to integrate such functionality with an external system.

Once the anagraphic data has been sent to the tool (from file if used as standalone, with a procedure call if used as a library), it will create a new document in the Certificate collection and return an image file with the corresponding QR code.

In future developments we leverage the possibility of enforcing security with an asymmetric encryption of the IDs (similarly to the systems implemented in many countries), that we will have to handle the codebases of both the mobile app and the qr code generator.

8 User guide

This final section contains some notes to get our system up and running.

8.1 Generating the data

Our example dataset can be obtained by simply running this script after cloning the relative repository.
Please note that, once run, the script will generate by default a new database on the remote cluster.
Alternatively, the user can specify the option "-local" as a command line argument to generate the data on a local mongoDB server.

8.2 Generating the QR codes

Without the need to write any code, one can test the generator by running it as a standalone. In order to insert some meaningful data in the database, the path to a file with some anagraphic content must be passed to the script as a command line argument. A template can be found in the repository.
The resulting QR code will be uploaded in the current working directory.

8.3 Testing the mobile application

As mentioned in the previous section, we delivered the entire codebase of the mobile application. However, Android users have the possibility to try out the prototype by installing the APK available on the repo, without the need of building the entire application.
A brief video demo is also available in the delivery folder.

9 Extra

A few days before the delivery, the Italian authorities addressed the problem of finding a technical way to revoke health certificates if the subject turns out to be positive. Such functionality (as described in the news) is actually already covered by our system, which takes into consideration all the relevant history of the subject at validation time. As described in the previous sections, the validity of a certificate can be later restored if the subject is proved to be healthy again.
We consider this to be a relevant proof of the robustness of our model.