Information System Project

Clinic Examinations Management System

Università di Pisa

Nicola Mota, Giacomo Pellicci, Daniele Nicolai, Bruk Tekalgne

November, 2019

Clinic Examinations Management System

A medical laboratory company have many offices all around the country.It needs a system so that doctors can share examination results quickly with their patients when they become available. The application will allow to keep all examination results for each patient in the clinic, and each doctor will be able to consult patient's examination and all the examinations they have carried out.

For this purpose, patients are required to register to the service by providing their personal information like name, surname, email so that they can consult all their examinations to know the results. Especially when a patient performs an examination, the doctor responsible for the test will record information about it, including the type and date. Doctors should be able to observe all examinations performed by themselves and by a specific patient. Once the examination result is available, the doctor can update the examination result and make it available to the patient for consultation. The result can be positive or negative. Only doctors are allowed to insert and update examinations records. Patients can only see their examinations. Doctors and patients can change their personal information within the system.

A patient may decide to delete past examinations that he carried out. Moreover, a patient may decide to remove all his/her personal information and all examinations performed within the clinic by cancelling their registration to the service.

Requirement Analysis

For the specified system we have the following functional and non functional requirements.

Functional Requirements

Here are listed the main functionalities that the system has to provide

- The system shall allow patients to sign up by providing name, surname, birth day, city of residence, sex, email and password
- The system shall allow patients to log in providing tax code and password
- The system shall allow doctors to log in providing id and password
- The system shall allow patients to modify their personal information: city of residence, email and password.
- The system shall allow doctors to modify their personal information: email and password.
- The system shall allow patients to consult their personal examination results.
- The system shall allow doctors to enter a new record for the patient's examination.
- An examination record shall include: type, date and result.
- The result shall have the possible value: not available, positive, negative.
- The system shall allow doctors to consult all examinations performed by a patient
- The system shall allow doctors to consult all examinations performed by themselves
- The system shall allow doctors to update the examination result.
- The system shall allow patients to remove past examinations records.
- The system shall allow patients to remove their accounts from the clinic.

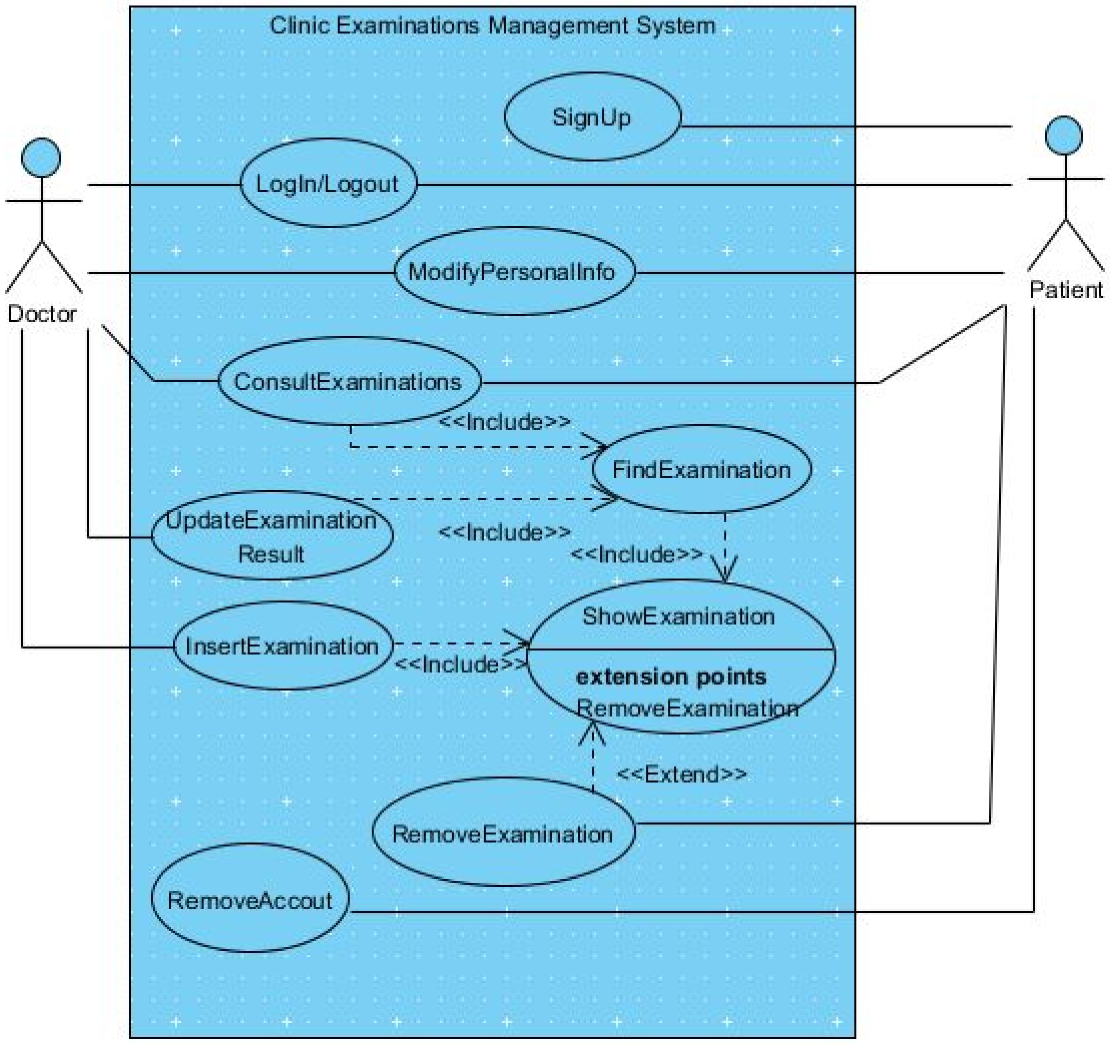
Nonfunctional Requirements

Here are the nonfunctional requirements of the system

- The system shall store data consistently
- The system shall allow short response times
- The system shall keep the user data privacy.
- The system shall store the passwords in a secure way.

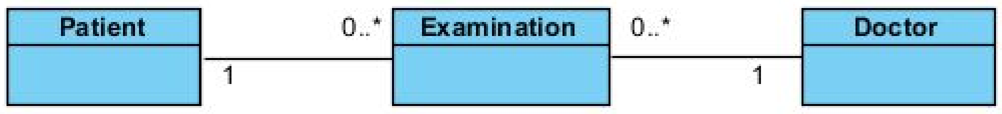
Use Case Diagram

The two main actors in our system are doctors and patients. Here we highlight the main use cases that can be performed by each actor.



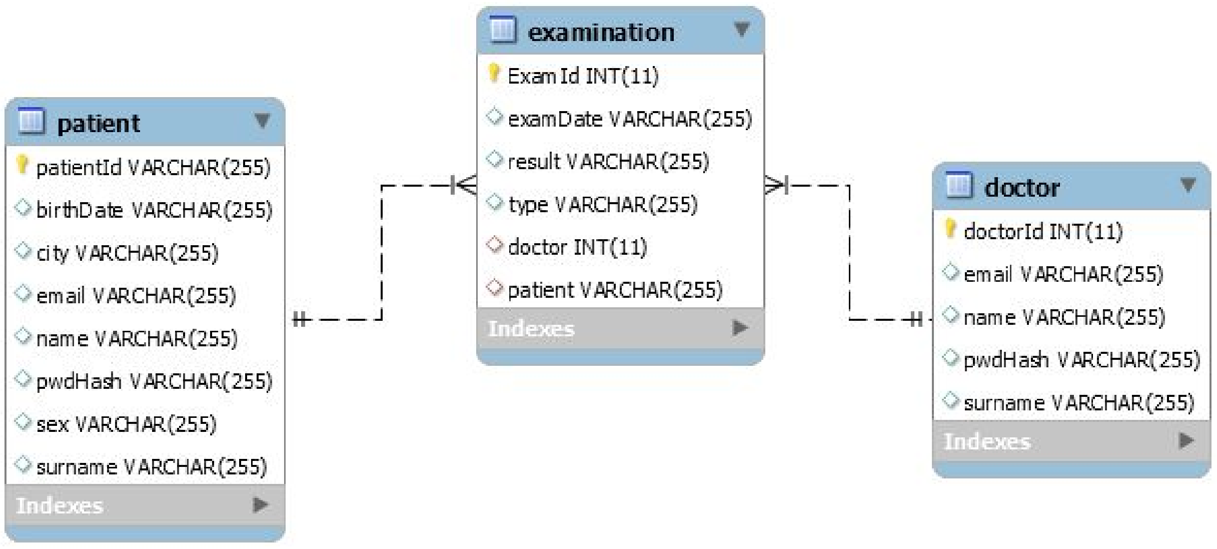
Database design

In the domain of the application, the main data entities are patient, examination and doctor. One patient can carry out many examinations, and one examination is related to only one patient. The same for a doctor that can perform many examinations each of which is related to him. So we have two 1-to-many associations that link our main data entities.



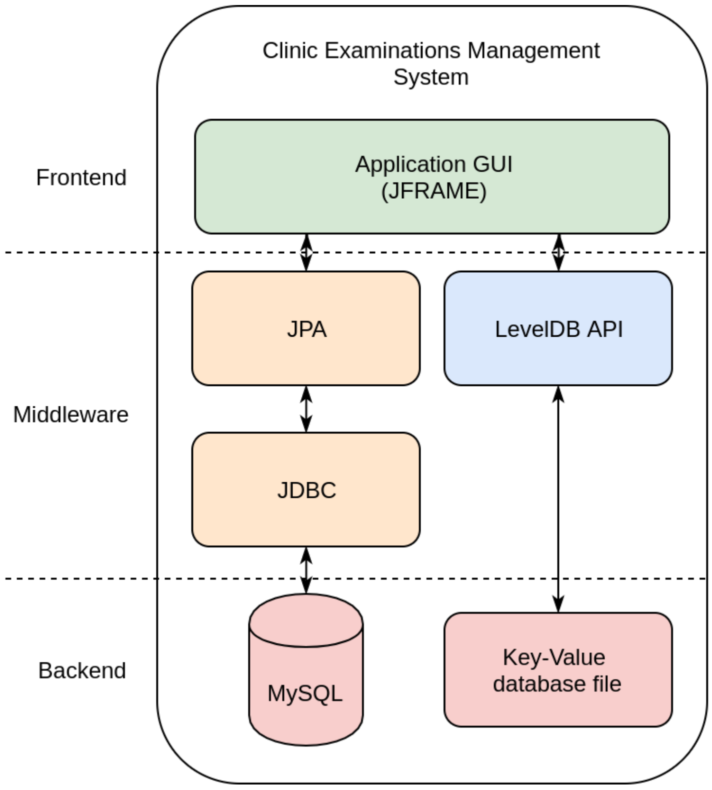
For each of these main entities we will have a 1 to 1 association with the entities that express each attribute. For the patient we will have a 1 to 1 association with name, surname, sex, address, e-mail, password and tax code. For the doctor with name, surname, email and password. And for the examination an association from 1 to 1 with type, date and result.

In order to handle this data model with a **relational database,** we obtain the structure for the database shown in the following ER diagram. The examination table implements the two 1 to many associations described before between patient and doctor entities with the foreign key represented by the two primary keys of these, in order to preserve referential integrity. Then we have a primary key **ExamId** and the main data attributes of the examination entity. Similarly for patient and doctor we have the primary keys, respectively **patientId** and **doctorId**, along with their attribute fields.



Software architecture

The system provides a GUI developed with JFrame for an easier interaction with doctors and patients. The information stored in the database will be accessed exploiting the Java Persistence API, which through JDBC will provide an object oriented access to data stored in the MySQL database. A further extension of the system will consist, after a feasibility study, in implementing a portion of the database as a non-relational one, in the form of key-value and that will be done using LevelDB APIs to access the key-value database.



Implementation

The application is developed in Java using Hibernate JPA for the implementation of the object relational mapping between the main data object and the database. We use JDBC connector to communicate with MySQL RDBMS. The GUI is developed using JFrame. The project dependencies are handled with Maven.

Java Persistence Api

Java Persistence Api is a framework for Java language that allows you to manage the persistence of data objects on relational databases. This allows to define an object relational mapping, i.e. a mapping between the data entities of object oriented languages and the structure of the database. The class is mapped in the table concept. An object of that class in a tuple of the table. A member field in an attribute. Trying to map concepts of object oriented languages such as the concept of encapsulation, scope, inheritance in relational databases is a problem.

JPA allows to mitigate the problems of the object relational impedance mismatch by favoring a logical link between the object oriented development and the relational structure.

JPA also defines an object oriented query language.

In order to use the framework it has been added to the Maven POM file along with the JDBC driver.

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.4.4.Final</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.17</version>
    </dependency>

    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.1.7</version>
    </dependency>

    <dependency>
        <groupId>org.iq80.leveldb</groupId>
        <artifactId>leveldb-api</artifactId>
        <version>0.12</version>
    </dependency>

    <dependency>
        <groupId>org.iq80.leveldb</groupId>
        <artifactId>leveldb</artifactId>
        <version>0.12</version>
    </dependency>

</dependencies>
```

The functionalities are provided in the javax.persistence package.

JPA Entities

To implement ORM the language defines annotations.

Here are listed the main annotations used in our code.

- The **@Entity** annotation is used to specify that the currently annotate class represents an entity type, describing the mapping between the actual persistable object of that class and a database table row.

- The **@Table** annotation is used to specify the primary table of the currently annotated entity. We have to specify the name of the table in the db if it is different from the name of the entity

- The **@Column** annotation is used to specify the mapping between a basic entity attribute and the database table column. We have to specify the name of the column in the table if it is different.

- The **@Id** annotation specifies the entity identifier. It models the primary key of an entity. Values for identifiers can be generated. To denote that an identifier attribute is generated, it is annotated with **@GeneratedValue.**

- The **@OneToMany** annotation is used to specify a one-to-many database relationship. If there is a **@ManyToOne** association on the child side, the **@OneToMany** association is bidirectional and the application developer can navigate this relationship from both ends. Although the Domain Model exposes two sides to navigate this association, behind the scenes, the relational database has only one foreign key for this relationship. Every bidirectional association must have one owning side only (the child side), the other one being referred to as the inverse (or **the mappedBy**) side. Unlike the unidirectional **@OneToMany**, the bidirectional association is much more efficient when managing the collection persistence state. Every element removal only requires a single update, and, if the child entity lifecycle is bound to its owning parent so that the child cannot exist without its parent, then we can annotate the association with the **orphan-removal** attribute and dissociate the child will trigger a delete statement on the actual child table row as well.

- The **@ManyToOne** annotation is used to specify a many-to-one database relationship. The **@JoinColumn** annotation is used to specify the FOREIGN KEY column used when joining an entity association

Following are the definition of our persistent objects, respectively Patient, Doctor and Examination:

Patient entity

```
@Entity
@Table(name = "patient")
public class Patient implements Serializable{
      @Column(name = "patientId")
      @Id
      private String patientId;
      private String name;
      private String surname;
      private String sex;
      private String birthDate;
      private String city;
      private String email;
      private String pwdHash;

      @OneToMany(mappedBy = "patient", orphanRemoval = true)
      private final List<Examination> examinations = new ArrayList<>();
}
```

Doctor entity

```
@Entity
@Table(name = "doctor")
public class Doctor implements Serializable {
      @Column(name = "doctorId")
      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      private int doctorId;
      private String name;
      private String surname;
      private String email;
      private String pwdHash;

      @OneToMany(mappedBy = "doctor", orphanRemoval = true)
      private final List<Examination> examinations = new ArrayList<>();
}
```

Examination entity

```
@Entity
@Table(name = "examination")
public class Examination implements Serializable  {
    @Column(name = "ExamId")
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int examinationId;

    @ManyToOne()
    @JoinColumn(name = "patient")
    private Patient patient;

    @ManyToOne()
    @JoinColumn(name = "doctor")
    private Doctor doctor;
    private String examDate;
    private String type;
    private String result;

}
```

ClinicManagerEM Class

ClinicManagerEM class manages the interaction with the database using the JPA framework.

EntityManager implements the programming interface defined by the JPA specification to manage entity instances defined in a persistence context.

```
public class ClinicManagerEM {

      private EntityManagerFactory factory;
      private EntityManager entityManager;
      //-----------------UTILITY METHODS
      public void setup() {
            factory = Persistence.createEntityManagerFactory("clinic");
      }

      public void exit() {
            factory.close();
      }
}
```

An EntityManagerFactory is used to create an EntityManager.

The EntityManagerFactory is obtained by the Persistence class specifying "clinic" as our persistence unit.

The configuration of a persistence unit is specified by a **persistence.xml** file that reside in the META-INF folder.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
```

```xml
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="clinic">
    <properties>
      <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/clinic?serverTimezone=UTC"/>
      <property name="javax.persistence.jdbc.user" value="xxxxx"/>
      <property name="javax.persistence.jdbc.password" value="xxxxxx"/>
      <property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="false"/>
      <property name="hibernate.format_sql" value="false"/>
      <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL8Dialect"/>
    </properties>
  </persistence-unit>
</persistence>
```

In properties we specify the connection parameters

- **javax.persistence.jdbc.driver**: the fully qualified class name of the driver class
- **javax.persistence.jdbc.url**: the driver specific URL
- **javax.persistence.jdbc.user**: the user name used for the database connection
- **javax.persistence.jdbc.password**: the password used for the database connection

and hibernate configuration parameters values

- **hibernate.hbm2ddl.auto**: set to value **update**.If a table doesn't exist then it creates new tables and where as if a column doesn't exist it creates new columns for it.

and to disable the SQL statements logging in console

- **hibernate.show_sql**: set to false
- **hibernate.format_sql**: set to false

ClinicManagerEM class also defines a set of CRUD operations required to implement the functionalities of our application.

Each operation is performed by creating an EntityManager from the factory.

To demarcate transaction boundaries we get an EntityTransaction through entityManager.getTransaction().Among other methods, the EntityTransaction API provides the **begin()** method to create a new transaction to the database and **commit()** to synchronize all changes made over Hibernate objects with the database, as well as committing those changes. In case the commit() operation fails the programmer can instead call the rollback() method.

EntityManager offers also a set of methods to manipulate our entities:

- **persist()** : is used to make an entity persistent in the database
- **find()**: retrieves an entity from the database by its primary key
- **getReference()**: retrieves a proxy object for an entity. It's a lightweight method to verify the existence of an entity when you don't have to access its field.
- **remove()**: removes a persistent entity from the database
- **merge()**:merges modifications made to an instance into the corresponding persistent instance, if one is already present. The merge operation automatically detect whether the merging of the instance has to result in an insert or update.
- **close()**:closes an entity manager instance releasing the resources associated to the transaction

Create operations

CreatePatient

This method implements the signup functionality required by our application storing persistently the information provided by the patient.

```
public void createPatient(String name, String surname, String sex, String
city, String birthDate, String email, String taxCode, String pwd) {

        Patient patient = new Patient();
        patient.setSurname(surname);
        patient.setName(name);
        patient.setBirthDate(birthDate);
        patient.setCity(city);
        patient.setEmail(email);
        patient.setSex(sex);
        patient.setPatientId(taxCode);
        patient.setPwdHash(Hash.getSHA256(pwd));

        try {
            entityManager = factory.createEntityManager();
            entityManager.getTransaction().begin();

            entityManager.persist(patient);

            entityManager.getTransaction().commit();
        } catch (RollbackException e) {
            System.out.println("rollback exception (duplicate)");
            entityManager.getTransaction().rollback();
        } catch (Exception e) {
            System.out.println("Exception in createPatient");
            entityManager.getTransaction().rollback();
        } finally {
            entityManager.close();
        }
    }
```

CreateExamination

The createExamination method is used to implement the functionality of the creation of a new examination record that is provided to doctors by storing its related information into the database.

```
public void createExamination(String patientId, int doctorId, String type,
String result, String examDate) {

        Examination examination = new Examination();
        examination.setType(type);
        examination.setDate(examDate);
        examination.setResult(result);
        try {
            entityManager = factory.createEntityManager();
            entityManager.getTransaction().begin();

            //find the patient
            Patient patient = entityManager.getReference(Patient.class,
patientId);
            if (patient == null) {
                throw new EntityNotFoundException();
            }
            examination.setPatient(patient);

            //find the doctor
            Doctor doctor = entityManager.getReference(Doctor.class,
doctorId);
            if (doctor == null) {
                throw new EntityNotFoundException();
            }
            examination.setDoctor(doctor);

            entityManager.persist(examination);
            entityManager.getTransaction().commit();  //commit
        } catch (EntityNotFoundException e) {
            System.out.println("createExamination - Entity not found");
            entityManager.getTransaction().rollback();
        } catch (EntityExistsException e) {
            System.out.println("createExamination - Entity already exists");
            entityManager.getTransaction().rollback();
        } catch (Exception e) {
            System.out.println("Exception in createExamination");
            entityManager.getTransaction().rollback();
        } finally {
            entityManager.close();
        }
```

```
    }
```

Read operations

ReadPatient

The readPatient method allow to retrieve the information of a patient record from the database.This function is used to implement the patient login operation.

```
public Patient readPatient(String taxCode) {
        Patient p = null;
        try {
            entityManager = factory.createEntityManager();
            //find that patient
            p = entityManager.find(Patient.class, taxCode);
            if (p == null) {
                throw new EntityNotFoundException();
            }
        } catch (EntityNotFoundException e) {
            System.out.println("readPatient -  taxCode not found");
        } catch (Exception e) {
            System.out.println("Exception in readPatient");
        } finally {
            entityManager.close();
            return p;
        }
    }
```

a similar readDoctor function is implemented to perform the same kind of operations for the doctor entity

ReadPatientExaminations

The method retrieve from the database the list of all examinations performed by the patient with a specific tax code

```
public List<Examination> readPatientExaminations(String taxCode) {
        Patient p = null;
        List<Examination> list = null;
        try {
            entityManager = factory.createEntityManager();

            //find that patient
            p = entityManager.find(Patient.class, taxCode);
            if (p == null) {
                throw new EntityNotFoundException();
            }
            else{
                list = p.getExaminations();
            }
        } catch (EntityNotFoundException e) {
            System.out.println("ReadPatientExaminations -  taxCode not 
found");
        } catch (Exception e) {
            System.out.println("Exception in readPatientExaminations");
        } finally {
            entityManager.close();
            return list;
        }
    }
```

ReadDoctorExaminations

Similarly to the readPatientExaminations, this method return the list of examinations performed by a doctor given the doctor id. This completes the two kind of search operation available for a doctor.

```
public List<Examination> readDoctorExaminations(int doctorId) {
        Doctor d = null;
        List<Examination> list = null;
        try {
            entityManager = factory.createEntityManager();

            d = entityManager.find(Doctor.class, doctorId);
            if (d == null) {
                throw new EntityNotFoundException();
            }
            else{
                list = d.getExaminations();
            }
        } catch (EntityNotFoundException e) {
            System.out.println("ReadDoctorExaminations -  not found");
        } catch (Exception e) {
            System.out.println("Exception in readDoctorExaminations");
        } finally {
            entityManager.close();
            return list;
        }
```

Update operations

UpdatePatientInfo

The updatePatientInfo performs the update operation over a patient record when he wants to change his own information regarding city, email and password.

```
public void updatePatientInfo(String taxCode, String city, String email,
String pwd) {
        if (city == null && email == null && pwd == null){
            return;
        }
        try {
            entityManager = factory.createEntityManager();
            entityManager.getTransaction().begin();

            //find that patient
            Patient p = entityManager.getReference(Patient.class, taxCode);
            if (p != null) {
                //able to update also 1 single field if you leave the other
blank
                if (city != null)
                    p.setCity(city);                        //update city
                if (email != null)
                    p.setEmail(email);                      //update email
                if (pwd != null)
                    p.setPwdHash(Hash.getSHA256(pwd));            //update pwd
hash

                entityManager.merge(p);                     //merge it

            } else {
                throw new EntityNotFoundException();
            }
            entityManager.getTransaction().commit();        //commit
        } catch (EntityNotFoundException e) {
            System.out.println("updatePatientInfo - taxCode not found");
            entityManager.getTransaction().rollback();
        } catch (Exception e) {
            System.out.println("Exception in updatePatientInfo");
            entityManager.getTransaction().rollback();
        } finally {
            entityManager.close();
        }
    }
```

a similar function is implemented in order to update information of a doctor regarding his email and password

UpdateExamination

When the result of an examination is available this is updated using this method.

```
public void updateExamination(int examinationId, String result) {
        try {
            entityManager = factory.createEntityManager();
            entityManager.getTransaction().begin();
            Examination e = entityManager.getReference(Examination.class,
examinationId);
            if (e != null) {
                e.setResult(result);
                entityManager.merge(e);
            } else {
                throw new EntityNotFoundException();
            }
            entityManager.getTransaction().commit();
        } catch(EntityNotFoundException ex){
            System.out.println("Entity not found in updateExamination (id: 
"+examinationId+")");
            entityManager.getTransaction().rollback();
        } catch (Exception ex) {
            System.out.println("Exception in updateExamination");
            entityManager.getTransaction().rollback();
        } finally {
            entityManager.close();
        }
    }
```

Delete operations

In order to perform the deletion of a patient in the system we implement the deletePatient method.

According to our cascade policy this would delete also all the examinations performed by the patient.

DeletePatient

```
public void deletePatient(String taxCode) {
        //due to CASCADE property, when you delete a patient you also delete 
all the
        //examination he had
        try {
            entityManager = factory.createEntityManager();
            entityManager.getTransaction().begin();

            //find that patient
            Patient p = entityManager.getReference(Patient.class, taxCode);
            if (p != null) {
                entityManager.remove(p);
            } else {
                throw new EntityNotFoundException();
            }
            entityManager.getTransaction().commit();
        } catch (EntityNotFoundException e) {
            System.out.println("deletePatient -  not found");
            entityManager.getTransaction().rollback();
        } catch (Exception ex) {
            System.out.println("Exception in deletePatient");
            entityManager.getTransaction().rollback();
        } finally {
            entityManager.close();
        }
    }
```

DeleteExamination

According to our requirements a patient can delete a past examination he carried out. In order to provide this functionality we implement the deleteExamination method shown below here.

```
public void deleteExamination(int examinationId) {

        try {
            entityManager = factory.createEntityManager();
            entityManager.getTransaction().begin();


            Examination e = entityManager.getReference(Examination.class,
examinationId);
            if (e!=null) {
                entityManager.remove(e);
            } else {
                throw new EntityNotFoundException();
            }
            entityManager.getTransaction().commit();
        } catch (EntityNotFoundException ex) {
            System.out.println("deleteExamination - entity not found");
            entityManager.getTransaction().rollback();
        } catch (Exception ex) {
            System.out.println("Exception in deleteExam");
            entityManager.getTransaction().rollback();
        } finally {
            entityManager.close();
        }

    }
```

Key Value Considerations

So far we have considered a relational database to handle our main data entities.

Although,considering this application domain, the overall data organization and management provided by a relational structure could give benefits regarding the generation of periodical reports that requires complex queries over multiple entities, thanks to the sql support, switching to a key-value architecture for some kind of frequent operations may allow us to achieve better performances and may give advantages also in terms of scalability and availability thanks to its simple structure.

In fact considering our application, one of the main operation that is performed is the retrieval of examinations informations that requires a join operation between the patient, doctor and examination tables performed over the foreign keys.

This in our relational database is equivalent to executing each time the following sql query

```
SELECT  e.visitId,d.name ,d.surname,d.email,p.name,p.surname,
        p.email,p.taxCode,e.date,e.type,e.result

FROM    patient p
        inner join examination e
          on p.patientId = e.patient
        inner join doctor d
          on e.doctor = d.doctorId

WHERE   p.taxCode = x
```

The use of a key-value database to handle this operation could provide better performances due to fast reads in the key-value store since we can retrieve all the required information directly searching by key avoiding each time to join together all the three tables informations.

In could provide also more flexibility for further extensions since it does not define a rigid scheme regarding the kind of information to store for an examination and its simpler structure allow easier partitioning of the data.

Key-value database implementation with LevelDB

For the key-value database implementation we used levelDB. It stores keys and values in arbitrary byte arrays, and data is sorted by key. It supports batching writes, forward and backward iteration.

In key-value databases there are three fundamental operations: put, get, delete. Also in LevelDB we have those operations and by combining them we can code the application logic. Data is meant to be accessed by key and not by value. All key-value pairs are kept sorted inside the database, that in fact is a file. Sorting order can be override to accomplish a specific order other than the lexicographic one applied by default. In the following a sample database dump:

```
doctor:1:email                      irene.taylor@hospital.it
doctor:1:name                       Irene
doctor:1:surname                    Taylor
doctor:2:email                      lisa.oliver@hospital.it
doctor:2:name                       Lisa
doctor:2:surname                    Oliver
doctor:3:email                      tim.clarkson@hospital.it
doctor:3:name                       Tim
doctor:3:surname                    Clarkson
examination:1:mcl1:1:examDate       2019-11-05 06:53:27
examination:1:mcl1:1:result         not available
examination:1:mcl1:1:type           HIV
examination:1:mcl1:1:examDate       2019-07-03 16:23:27
examination:1:mcl1:1:result         negative
examination:1:mcl1:1:type           Hepatitis
examination:2:mcl1:1:examDate       2019-10-22 09:11:27
examination:2:mcl1:1:result         positive
examination:2:mcl1:1:type           Anemia
patient:bck1:email                  paul.buckland@skynet.com
patient:bck1:name                   Paul
patient:bck1:surname                Buckland
patient:clm1:email                  jennifer.coleman@gmail.com
patient:clm1:name                   Jennifer
patient:clm1:surname                Coleman
patient:mcl1:email                  austin.mclean@mymail.it
patient:mcl1:name                   Austin
patient:mcl1:surname                McLean
```

We can identify the key-value configuration in the database in 3 types:

```
doctor:$doctor_id:$attribute_name = $value
examination:$examination_id:$doctor_id:$patient_id:$attribute_name = $value
patient:$patient_id:$attribute_name = $value
```

This configuration allows us to access doctor and patient info by key and also read examinations of a given patient or doctor by key.

With that being said, here are the main operations carried out in the key-value implementation of the database:

Create methods

```java
public void putPatient(String name, String surname, String email, String
taxCode) {
      String key = "patient:" + taxCode + ":";
      String s = get(key+"name");
      if (s != null)
            return;
      put(key + "name", name);
      put(key + "surname", surname);
      put(key + "email", email);
}
```

This method put a patient into the key-value database. It adds a pair for each patient attribute (name, surname, email) only after a duplicate check. Also putDoctor() and putExamination() methods works in a similar manner.

Update operations

```java
public void updatePatientInfo(String taxCode, String email) {
      String key = "patient:" + taxCode + ":";
      if(email != null)
            put(key+"email", email);
}
```

Update methods are all structured in this way, we build the key and then we put the value we want to update in the database. For example, assuming you want to modify the email address of doctor number 1, you will access this key-pair putting the new email, making a transition in the database status from:

```
doctor:1:email                    irene.taylor@hospital.it
```

to:

```
doctor:1:email                    my.changed.mail@hospital.it
```

with the operation:

```
put("doctor:1:email", "mychangedmail@hospital.it")
```

Read operations

Simple reads are performed with LevelDB method get() inside readPatient() and readDoctor() methods. More complex methods like readPatientExamination and readDoctorExamination have a similar structure, here we report one:

```
 public List<Examination> readPatientExamination(String taxCode){
        String myKey = "examination:";
        DBIterator iterator = levelDBStore.iterator();
        iterator.seek(bytes(myKey)); // starts from the specified key
        ArrayList<String> param = new ArrayList<>();
        final List<Examination> examinations = new ArrayList<>();
        int i = 0;
        while (iterator.hasNext()){
              byte[] key = iterator.peekNext().getKey();
              String[] keySplit = asString(key).split(":"); // split the key
              if(!keySplit[0].equals("examination"))
                    break;
              if(keySplit[2].equals(taxCode)){
                    byte[] value = iterator.peekNext().getValue();
                    param.add(asString(value));
                    i++;
                    if(i==3){
                          Patient p = readPatient(keySplit[2]);
                          Doctor d = readDoctor(Integer.parseInt(keySplit[3]));
                          Examination e = new Examination(
                                Integer.parseInt(keySplit[1]), p, d,
                                param.get(0), param.get(2), param.get(1)
                          );
                          examinations.add(e);
                          param.clear();
                          i=0;
                    }
              }
              else{
                i = 0;
            }
            iterator.next();
        }
        return examinations;
}
```

In readPatientExamination() method we are looking for all examinations carried out by a certain patient. We first define an iterator that will be used to traverse the database. With seek() we start from the first key-value pairs whose key contains "examination:" just to avoid a full traverse of the database. Then at each iteration they key is split into pieces, one for every ":". Then when the patientId matches the proposed one we store the information needed in list. This list of examinations contains for each examination all the information that we need to show to the user when the method is invoked. Also a check must be performed, in order to reduce the portion of database traversed for the end of the "examination portion" of the database as done with:

```
if(!keySplit[0].equals("examination"))
        break;
```

Delete operation

Delete operation are carried out by invoking the delete() method, here we report the deletePatient() method code, which is almost self explanatory:

```
public void deletePatient(String taxCode) {
        String myKey = "patient:" + taxCode + ":";
        DBIterator iterator = levelDBStore.iterator();
        iterator.seek(bytes(myKey)); // starts from the specified key
        while (iterator.hasNext()){
            byte[] key = iterator.peekNext().getKey();
            String[] keySplit = asString(key).split(":"); // split the key
            if (!keySplit[0].equals("patient") ||
                !keySplit[1].equals(taxCode)) { // breaking condition
                    break;
            }
            delete(asString(key));
            iterator.next();
        }
}
```

We simply seek() to the first key-value pair where the key refers to the given patient and then we iterate until all the fields are deleted. Also here we use the breaking condition to avoid a full traverse from the seek point to the end of the database.

User Manual

At the startup the application shows the main window. This window provide an options to select from. Each user depending on his role clicks on the login button below it's icon e.g. A patient clicks below patient icon. There is also an option for sign up provided that the patient is a newcomer to the hospital. where it leads him to insert his information.

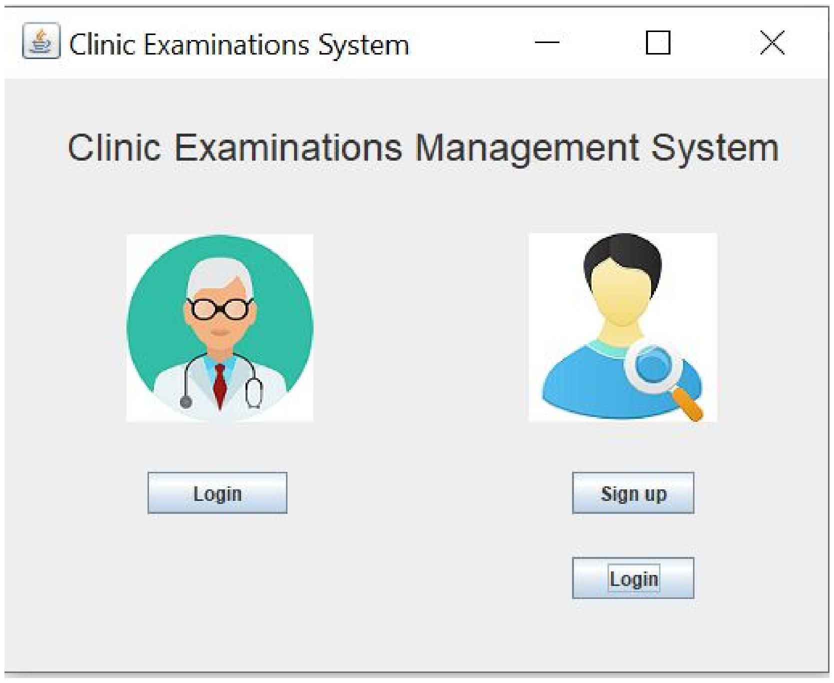

Clinic Examinations System
Clinic Examinations Management System
Login
Sign up
Login

Fig 1: Main screen of the Application

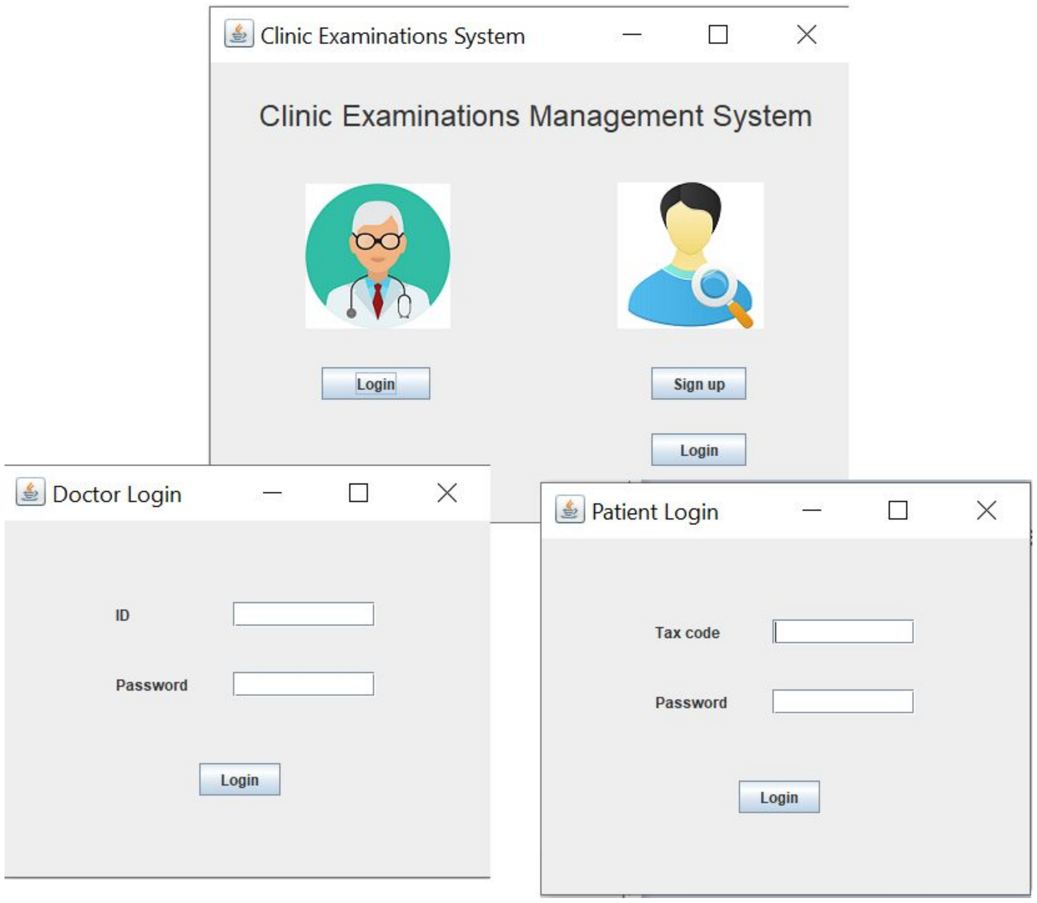
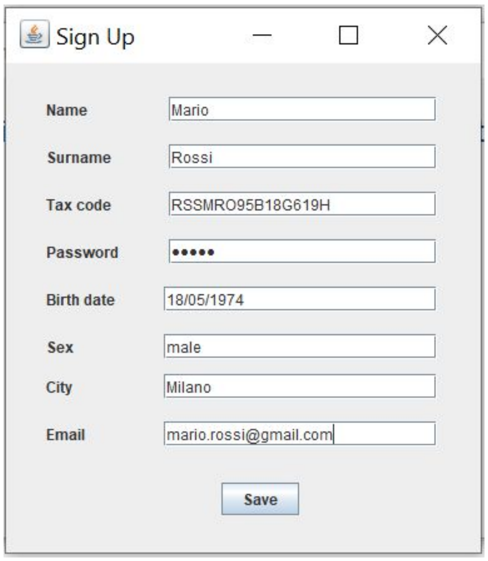


Fig 2: Login windows for users of the system

Patient Manual

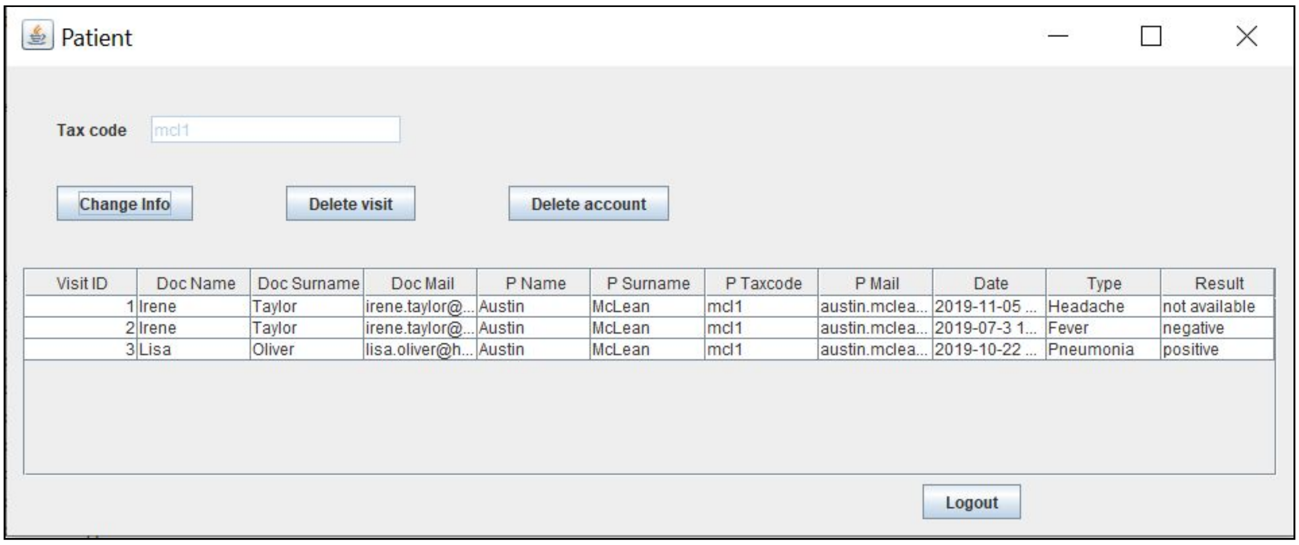
To **sign up** into the system, click **the sign up** button in the main window. In the Sign Up window. **Enter the personal information** required and click **save**. Now the patient is registered into the system and he can log in to use all the functionalities available.



Name	Mario
Surname	Rossi
Tax code	RSSMRO95B18G619H
Password	•••••
Birth date	18/05/1974
Sex	male
City	Milano
Email	mario.rossi@gmail.com

Fig 3: Sign Up window for patient

After inserting the correct credentials in the specified login window the patient is allowed to enter the system. He is immediately presented with his examination history with a window that is shown below.

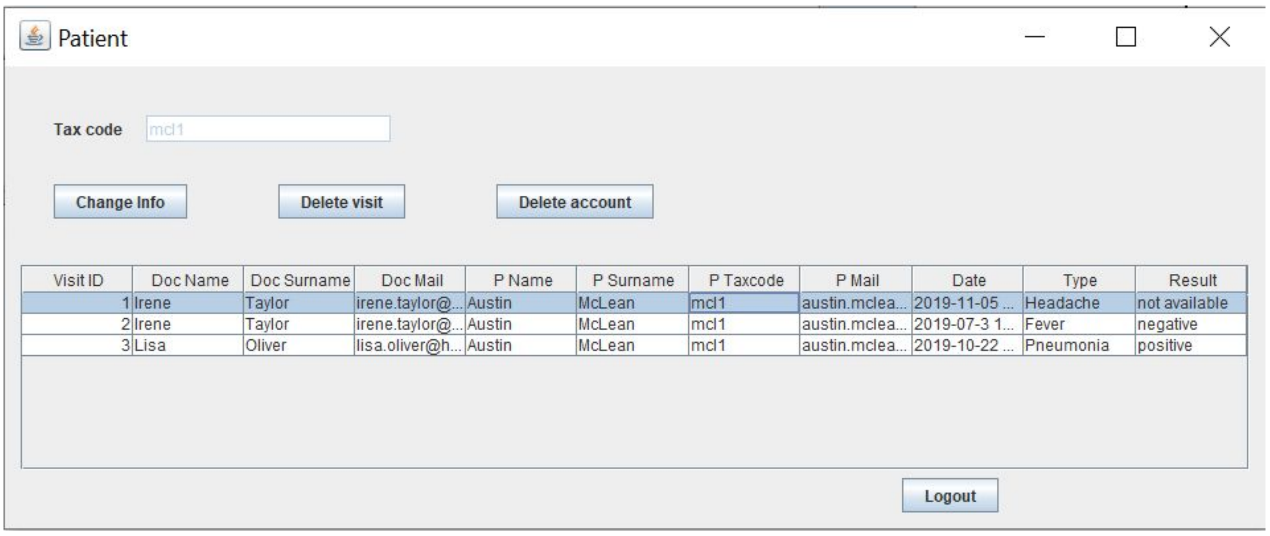


Tax code: mcl1

Visit ID	Doc Name	Doc Surname	Doc Mail	P Name	P Surname	P Taxcode	P Mail	Date	Type	Result
1	Irene	Taylor	irene.taylor@...	Austin	McLean	mcl1	austin.mclea...	2019-11-05 ...	Headache	not available
2	Irene	Taylor	irene.taylor@...	Austin	McLean	mcl1	austin.mclea...	2019-07-3 1...	Fever	negative
3	Lisa	Oliver	lisa.oliver@h...	Austin	McLean	mcl1	austin.mclea...	2019-10-22 ...	Pneumonia	positive

Fig 4: medical history of a patient

To **delete an examination, select in the table the examination** that you want to delete and click to **delete visit.**

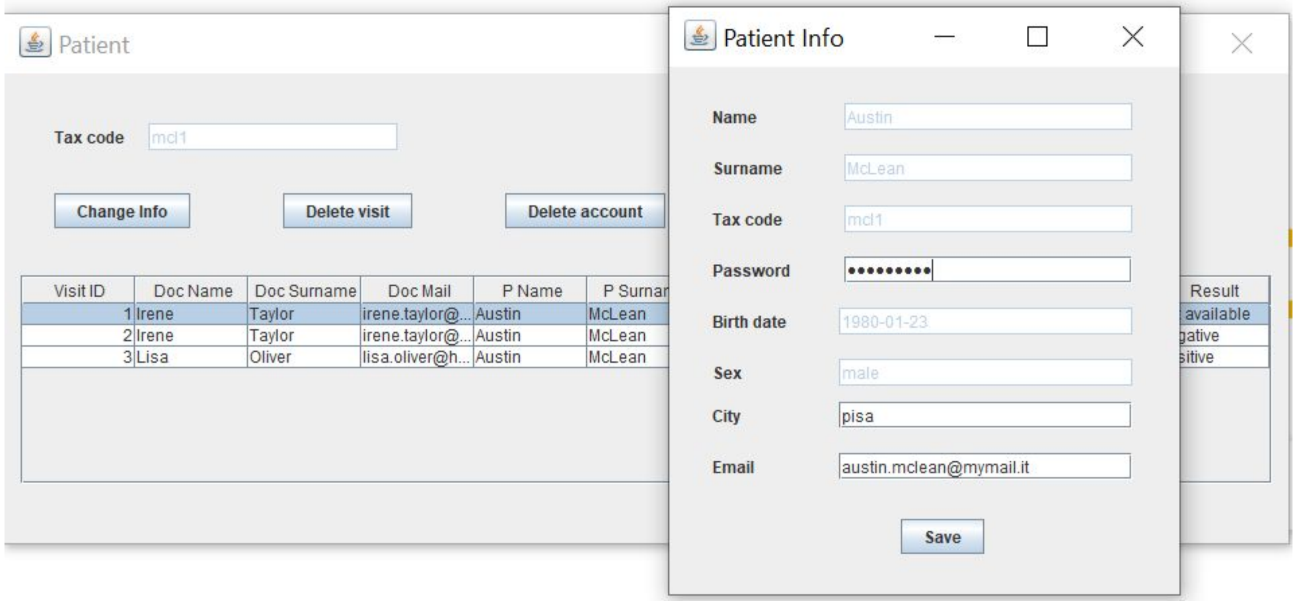

Patient

Tax code: mcl1

Visit ID	Doc Name	Doc Surname	Doc Mail	P Name	P Surname	P Taxcode	P Mail	Date	Type	Result
1	Irene	Taylor	irene.taylor@...	Austin	McLean	mcl1	austin.mclea...	2019-11-05 ...	Headache	not available
2	Irene	Taylor	irene.taylor@...	Austin	McLean	mcl1	austin.mclea...	2019-07-3 1...	Fever	negative
3	Lisa	Oliver	lisa.oliver@h...	Austin	McLean	mcl1	austin.mclea...	2019-10-22 ...	Pneumonia	positive

Fig 5: showing how a patient can select a single exam

To **change personal information**, click **change info**. In the new window that appears insert the updated information among: **email**, **city** and **password,** that you want to change and click **save**.


Patient Info

Name	Austin
Surname	McLean
Tax code	mcl1
Password	•••••••••
Birth date	1980-01-23
Sex	male
City	pisa
Email	austin.mclean@mymail.it

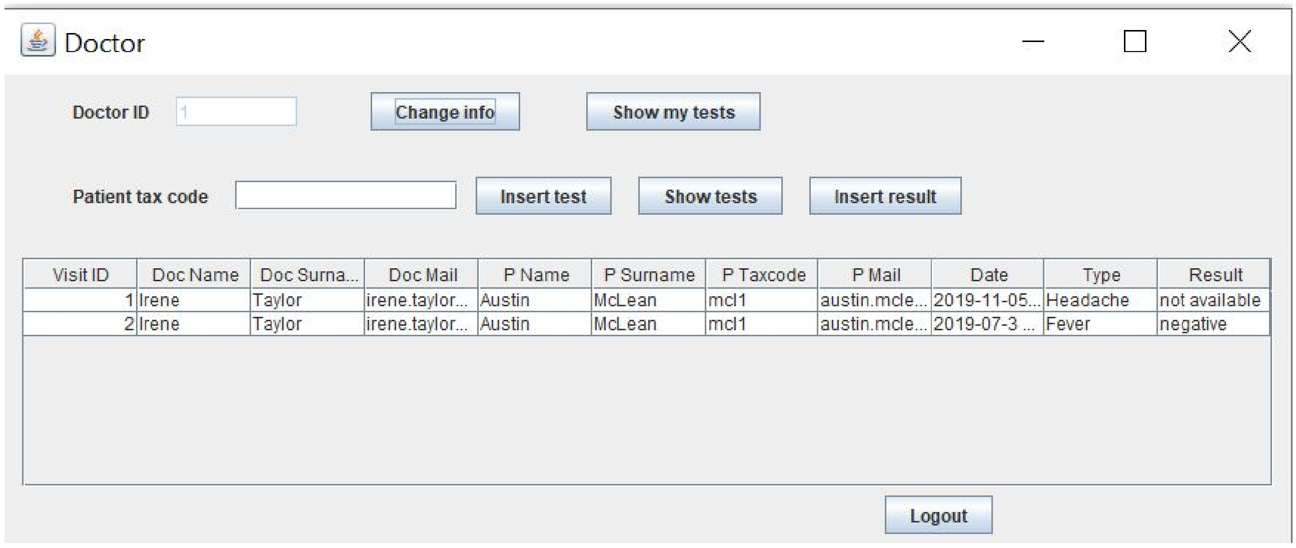
Fig 6: Updating patient's own personal information

To **remove one's own account** click on **delete account.** The system will remove all the information regarding the patient including his visits.

To **return to the main window** click **logout,** at the bottom.

Doctor Manual

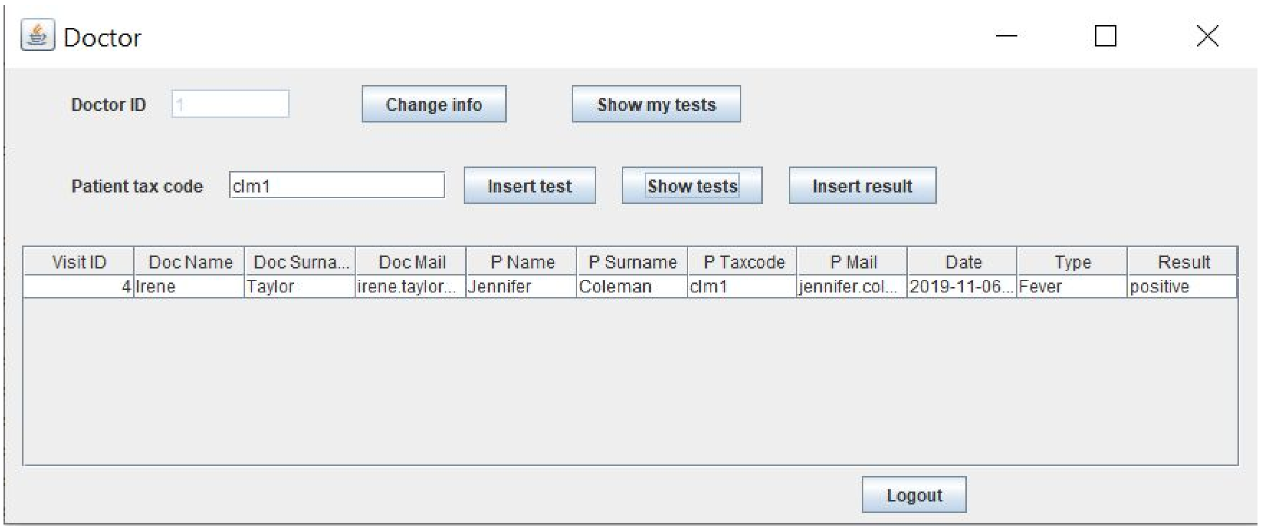
When a doctor logs in, he is led to the following window which will show him all the examinations performed by that doctor.



Visit ID	Doc Name	Doc Surna...	Doc Mail	P Name	P Surname	P Taxcode	P Mail	Date	Type	Result
1	Irene	Taylor	irene.taylor...	Austin	McLean	mcl1	austin.mcle...	2019-11-05...	Headache	not available
2	Irene	Taylor	irene.taylor...	Austin	McLean	mcl1	austin.mcle...	2019-07-3 ...	Fever	negative

Fig 7: Doctor's main interface

To **see the examinations of a certain patient** insert the **tax code** of the patient and click **show tests.** The system will show all the examinations of that patient performed by any doctor.

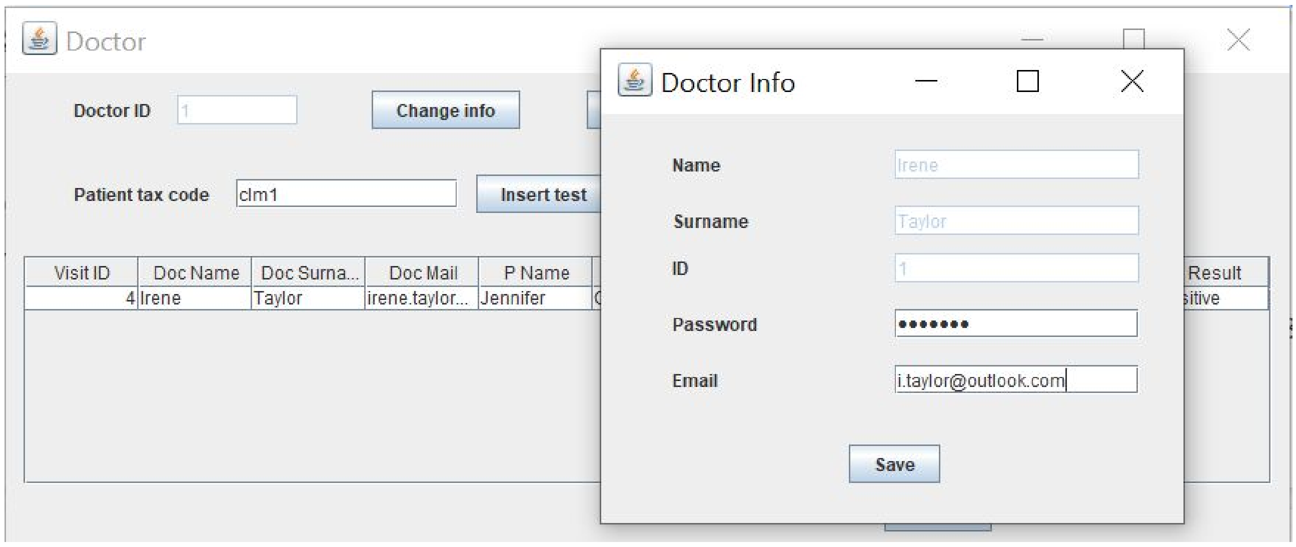


Visit ID	Doc Name	Doc Surna...	Doc Mail	P Name	P Surname	P Taxcode	P Mail	Date	Type	Result
4	Irene	Taylor	irene.taylor...	Jennifer	Coleman	clm1	jennifer.col...	2019-11-06...	Fever	positive

Fig 8: doctor observing examinations performed to a patient

To return to see all the examinations performed by you as a doctor, click **show my tests.**

To **change your personal information**, click **change info.** In the following window enter the new **email** or the new **password**, then click **save**.



Doctor ID 1

Patient tax code clm1

Visit ID	Doc Name	Doc Surna...	Doc Mail	P Name	Result
4	Irene	Taylor	irene.taylor...	Jennifer	sitive

Name Irene

Surname Taylor

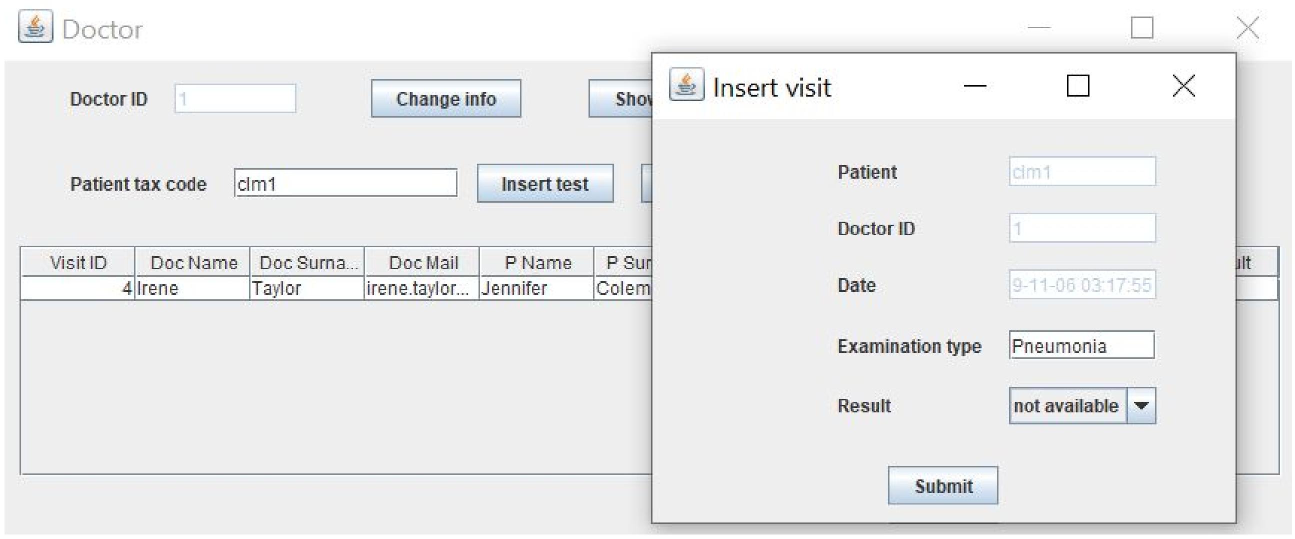
ID 1

Password •••••••

Email i.taylor@outlook.com

Fig 9: A doctor updating his personal information in the system

To **insert a new test,** enter the **patient tax code** then click **insert test**. In the window that appear enter the **type of the examination** and the **result** and press **submit.**



Doctor ID 1

Patient tax code clm1

Visit ID	Doc Name	Doc Surna...	Doc Mail	P Name	P Sur	lt
4	Irene	Taylor	irene.taylor...	Jennifer	Colem	

Patient clm1

Doctor ID 1

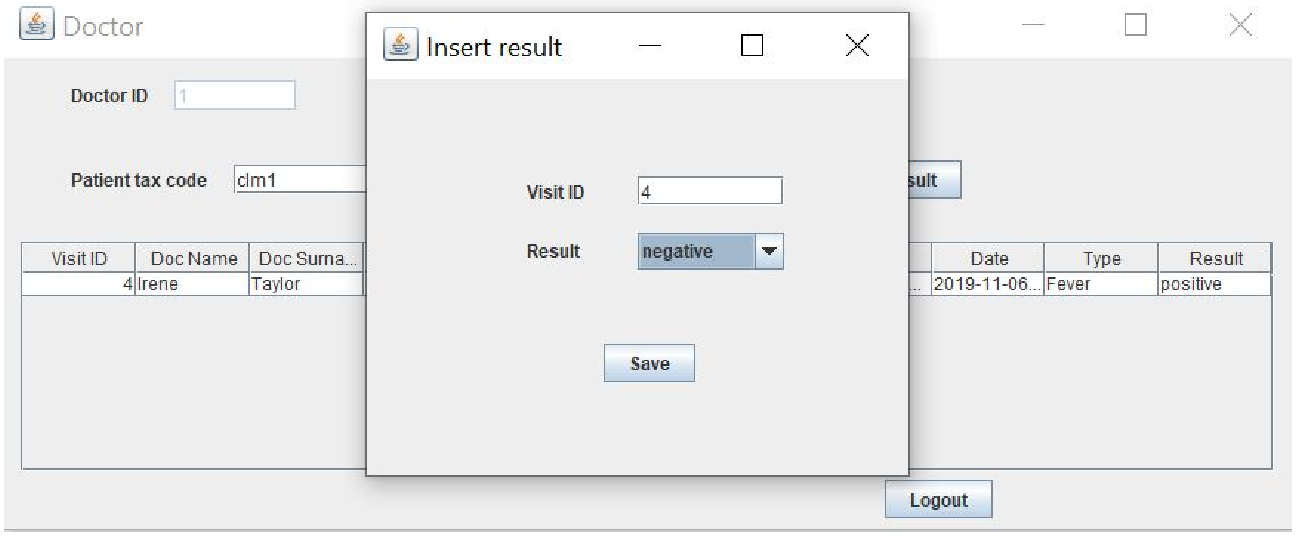
Date 9-11-06 03:17:55

Examination type Pneumonia

Result not available

Fig 10: A doctor inserting a new examination without result

To insert a result of a test, click **insert result**. In the new window insert the **visit id** and the **result** of the visit then click **save.** A doctor can also insert the result of the examination together with other examination details.



Doctor

Doctor ID: 1

Patient tax code: clm1

Insert result

Visit ID: 4

Result: negative

Visit ID	Doc Name	Doc Surna...	Date	Type	Result
4	Irene	Taylor	2019-11-06...	Fever	positive

Figure 11: A doctor inserting an exam result

To **return to the main window** click **logout.**