ESILV 2020/2021 **Nada NAHLE**
Java EE & Frameworks
Session 1 – Learning Java/Javafx
18/09/2020

TUTORIEL JAVAFX

The tutorial presents the basic steps to do in order to develop a simple CRUD application in JavaFX. This application allows us to Create, Retrieve, Update and Delete students from/to a Mysql database.
We have to follow the MVC design pattern by organizing our code into Model (Java classes), View (Fxml documents), and Controllers (Java classes).

Each student is described by the following:

- Id : Int
- Name : String
- Gender : String (Male/Female)
- Birth date : LocalDate
- Photo → String (URL)
- Mark → String
- Comments → String

Application features:

1. Retrieve the list of students from the database and display their names in a ListView.
2. Display the details of a student when his name is selected:
    - Name → TextField
    - Gender → ComboBox
    - Birth date → DatePicker
    - Photo → ImageView
    - Mark → TextField
    - Comments → TextArea
3. Edit the information of an existing student .
4. Add new student by filling at least the name and gender.
5. Delete a student.

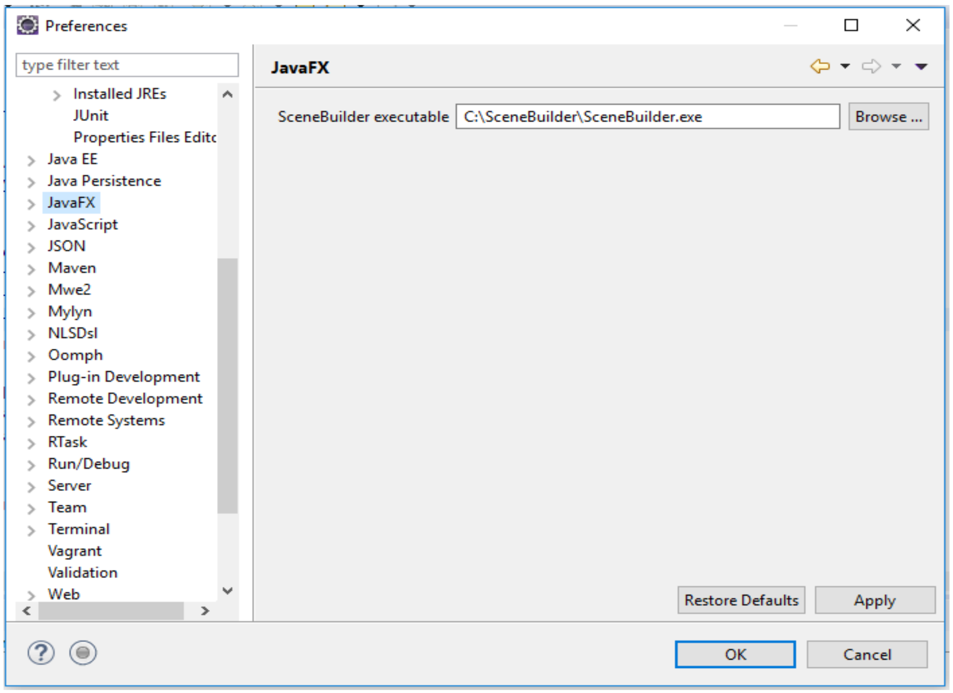
Software Requirements:

1. Java Developement Kit

2. Eclipse IDE

3. Javafx plugin in eclipse

4. SceneBuilder
   To connect Eclipse to SceneBuilder: go to the Window menu → Preferences → Javafx and Browse for the executable scenebuilder.



Preferences

JavaFX

SceneBuilder executable: C:\SceneBuilder\SceneBuilder.exe

5. MySQL

6. The JDBC driver : The connector/J jar file.
   This jar file used as external jar allows us to use mysql from eclipse.

Step1 : Create the project

Create a new JavaFX project → StudentFX

You have by default in your new project TutorielFX a source folder « src » and an « application » folder with a main class and an empty « application.css » for styling the interface.

Step2 : Create the Model

- Create a new java class in the « application » folder. This class should contain the model. Lets name it « Student ».
- Declare all student attributes.
- Generate the getters and setters for these attributes by doing a right-click on the code page → source → generate getters and setters → check the attributes → ok.
- Generate a constructor for the class by doing a right-click on the code page → source → generate constructor using fields → check the fields → ok.
- Write a constructor with just the mandatory fields (name and gender).

Step3 : Create the view

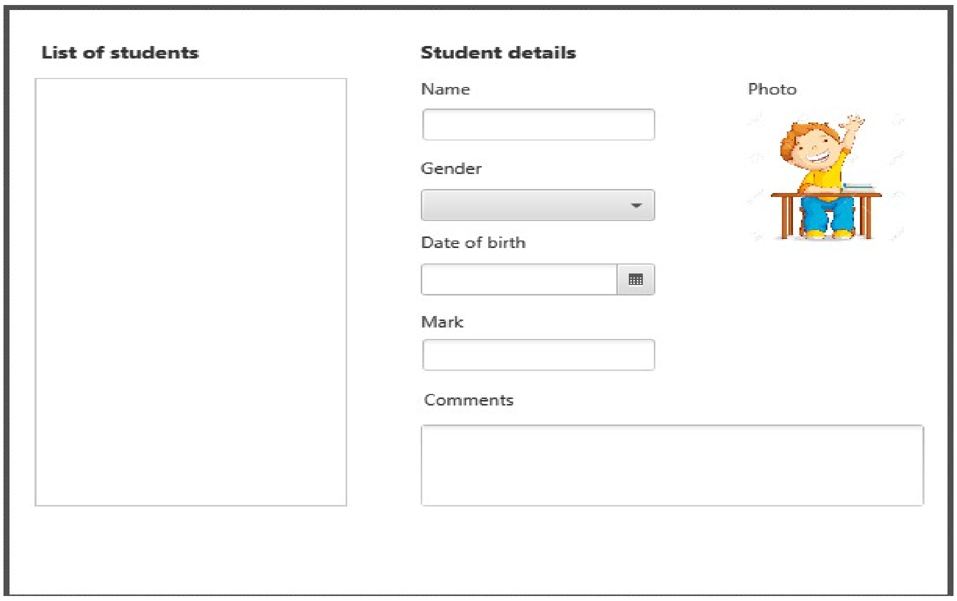
- Create a new fxml document by doing a right-click on the application folder → new → other → javafx → new fxml document. Lets name it « Student ». You can now right-click on the fxml file and open it with scenebuilder.

Step 4 : Building the interface

Delete the AnchorPane created by default. Drag a new one into the middle and put the needed controls in it:

- Some labels
- a ListView to list student names
- TextFields for name and mark
- ComboBox for gender
- DatePicker for date of birth
- TextArea for comments
- ImageView for the photo

The result should be like the following :



In the right side of your scenebuilder, you have a menu called « Code ». Give fx:id for each control. This id is what we will use later from the controller to do operations on the control.

When you give ids for all controls, save the interface in scenebuilder and close it. Refresh your project.
Open your Student.fxml and look to the code. It should be updated.

Step 5: Define gender values

In order to add the two values (male and female for gender), you can edit your fxml file by adding the following code in the ComboBox tag :

```
<ComboBox fx:id="cmbGender" layoutX="263.0" layoutY="140.0" prefWidth="150.0">
  <items>
      <FXCollections fx:factory="observableArrayList">
                   <String fx:value="Male" />
                   <String fx:value="female" />
      </FXCollections>
  </items>
</ComboBox>
```

and by adding the following imports at the top of the document :

```
<?import javafx.collections.FXCollections?>
<?import java.lang.String?>
```

Save your code and reopen « student.fxml » with scenebuilder. The gender contains now a drop down list with two values male and female.
N.B :These values can also be added dynamically from the controller. I

will show you how when we create the controller.

Step 6 : Creating the controller class

Create a new class in your application folder. Lets name it StudentController.

We have to establish the link between our view and our controller :

- We have to define each control of the view in the controller using the FXML annotation (@FXML). For example, the label lblName of student.fxml will be represented by the following in the controller (you must exactly the same name as in the view) :

  ```
  public class StudentController{

  @FXML
  Label lblName;

  @FXML
  Label lblGender;


  ...
  }
  ```

  Do that for all the controls of the view. Save the project.
- Open student.fxml with sceneBuilder. You can find on the left at the bottom a menu called « Controller ». Choose « StudentController. ».

Now, the view (Student.fxml) and the Controller (StudentController) are connected.

Step 7 (Optional): Define gender values dynamically

Since we have to define gender values when the form is initialized, we can choose that StudentController implements the « initializable » interface :

```
public class StudentController implements Initializable {
```

You will have an error, repair it by adding unimplemented methods. The « initialize » method appears. Put the following code inside :

```
List<String> gvalues = new ArrayList<String>();
gvalues.add("Male");
gvalues.add("Female");
ObservableList<String> gender = FXCollections.observableArrayList(gvalues);
cmbGender.setItems(gender);
```

Step 8: Editing the Main Class

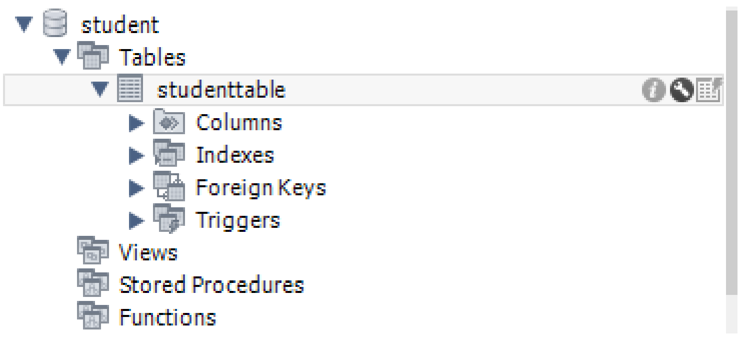
To open the application from the Student.fxml instead of an empty BorderPane, replace the line of parent root by the following :

```
Parent root = FXMLLoader.load(getClass().getResource("/application/Student.fxml"));
```

You can also set a title for the Stage and set the height and width of the form (scene).
Run your application. The form should appear with all the controls inside.
Verify the gender values.

Step 9: Creating the database

Open MySql Workbench. Create a new database (student) with one table (studenttable).



Then, define columns as following :

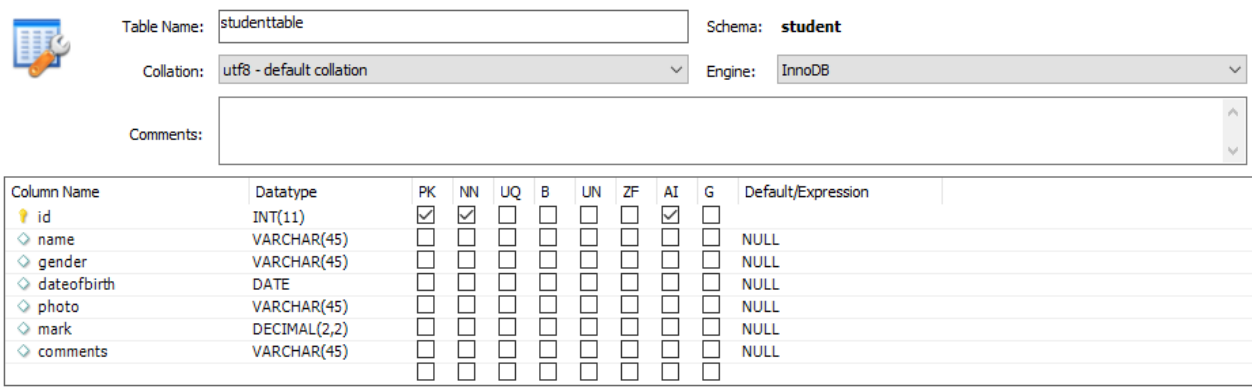
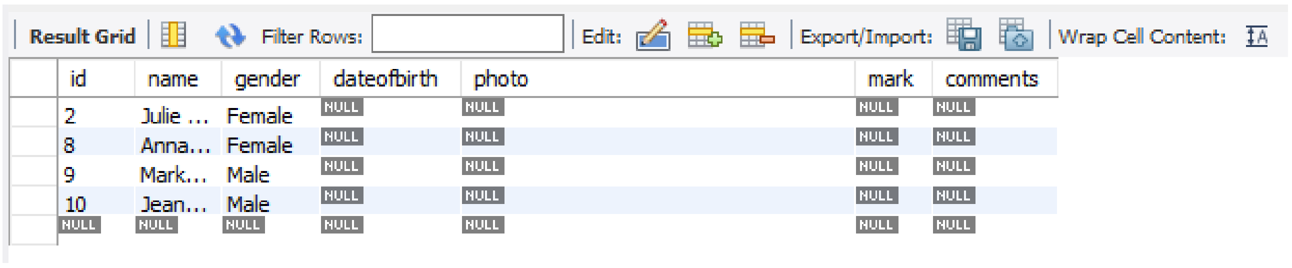


Table Name: studenttable Schema: **student**
Collation: utf8 - default collation Engine: InnoDB
Comments:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT(11)	✓	✓	☐	☐	☐	☐	✓	☐	
name	VARCHAR(45)	☐	☐	☐	☐	☐	☐	☐	☐	NULL
gender	VARCHAR(45)	☐	☐	☐	☐	☐	☐	☐	☐	NULL
dateofbirth	DATE	☐	☐	☐	☐	☐	☐	☐	☐	NULL
photo	VARCHAR(45)	☐	☐	☐	☐	☐	☐	☐	☐	NULL
mark	DECIMAL(2,2)	☐	☐	☐	☐	☐	☐	☐	☐	NULL
comments	VARCHAR(45)	☐	☐	☐	☐	☐	☐	☐	☐	NULL
		☐	☐	☐	☐	☐	☐	☐	☐	

Finally, fill some students with just name and gender. The « id » is auto-incremented.



id	name	gender	dateofbirth	photo	mark	comments
2	Julie ...	Female	NULL	NULL	NULL	NULL
8	Anna...	Female	NULL	NULL	NULL	NULL
9	Mark...	Male	NULL	NULL	NULL	NULL
10	Jean...	Male	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL

Step 10: Adding the JDBC driver

To add the JDBC driver to your project, right-click on your project → build path → configure build path → select Classpath → Add External Jars → put the path of your installed jbdc driver (see step 5). Verify with your projet explorer that the driver appears in the referenced libraries.

Step 11: Retrieving data

Now, we have all what we need to fetch student names from the database and display them in the listview. Let us describe the operation :
Once the application is launched, the listview should appear directly with the student names. So, the work should be done when the view is initialized. The starting point is then the controller of student.fxml ; StudentController.

We have already implemented the initialize method in StudentController. We will add a call for a method « fetchstudents » that we will write separately in studentController.

```
@Override
public void initialize(URL location, ResourceBundle resources) {

	List<String> gvalues = new ArrayList<String>();
	gvalues.add("Male");
	gvalues.add("Female");
	ObservableList<String> gender = FXCollections.observableArrayList(gvalues);
	cmbGender.setItems(gender);

	fetchStudents();
}
```

The method fetchstudents should communicate with the database.
A good practice is to delegate this work to a new class dedicated for database management. This class is a part of the model.
Create a new class « DBManager ». In this class, we will get a connection to the database and do all the manipulations on students.

Define an attribute of type DBManager in StudentController and create an instance on initialiazation.

```
DBManager manager;
@Override
public void initialize(URL location, ResourceBundle resources) {
                  manager = new DBManager();
…}
```

The method « fetchstudents » in StudentController uses a method « loadStudents » of DBManager as a black box. « LoadStudents » returns a list of student names. Fetchstudents affects this list to the listview as following :

```
public void fetchStudents() {
      ObservableList<String> students;
      if (manager.loadStudents()!=null) {
            students= FXCollections.observableArrayList(manager.loadStudents());
            lvStudents.setItems(students);
            }

      }
```

Now, we will write the method « loadStudents » in DBManager.

```
public class DBManager{

      public List<String> loadStudents(){
            List<String> studentNames= new ArrayList<String>();
            Connection myConn= this.Connector();
            try {
                  Statement myStmt= myConn.createStatement();
                  String sql = "select name from studenttable";
                  ResultSet myRs= myStmt.executeQuery(sql);
                  while (myRs.next()) {
                        String name= myRs.getString("name");
                        studentNames.add(name);
                  }
                  this.close(myConn, myStmt, myRs);
                  return studentNames;

            } catch (SQLException e) {
                  e.printStackTrace();
            }
            return null;
      }


      public Connection Connector(){
            try {
                  Connection connection = 
                  DriverManager.getConnection("jdbc:mysql://localhost:3306/student",
                  "username","password");
                  return connection;
```

```
			}
			
			catch (Exception e) {
				e.printStackTrace();
				return null;
			}
		}
		
		private void close(Connection myConn, Statement myStmt, ResultSet myRs) {
		
			try{
				if(myStmt!=null)
					myStmt.close();
				if(myRs!=null)
					myRs.close();
				if(myConn!=null)
					myConn.close();
			}
			catch(Exception e){
				System.out.println(e.getMessage());
			}
		}
```

Note that the method loadStudents needs to get a connection and close it after the operation. Since we have to repeat these actions for every operation, we can write them in two separate methods « Connector » and « close » .
Run your application. The listview should be filled with student names.

Step 12: Displaying a choosed student data

Now, once we have the list of names, we should be able to select a student and display the information about this student in the corresponding controls (name and gender for the moment).
So, we need an event listener on the listview. We add it in the initialize method of StudentController :

```
lvStudents.getSelectionModel().selectedItemProperty().addListener(e->
displayStudentDetails(lvStudents.getSelectionModel().getSelectedItem()));
```

Note that this event listener declaration is done using lambda expression, one of the most useful features of Java 8.

```
e-> displayStudentDetails(lvStudents.getSelectionModel().getSelectedItem())
```

Here, the event e executes the method displayStudentDetails, that we have to write in the next step. This method will also delegate the retrieve of student information to the class DBManager. Once the student is retrieved, it displays his info in the corresponding controls.

```
private void displayStudentDetails(String name) {

        try{
        Student s =manager.fetchStudentByName(name);

        txtName.setText(s.getName());

        cmbGender.setValue(s.getGender());
        dpBirth.setValue(s.getDob());

        Image image;
        if(s.getPhoto()!=null) {
        File file= new File(s.getPhoto());
                image= new Image(file.toURI().toString());
                ivPhoto.setImage(image);
        }
        else {
                image= new Image ("/picture/student.jpg");
                ivPhoto.setImage(image);
        }
        txtMark.setText(String.valueOf(s.getMark()));
        txtComments.setText(s.getComments());
        }
        catch (Exception e) {
        System.out.println(e.getMessage());
        }
}
```

And here is the code of « fetchstudentbyname » in DBManager.

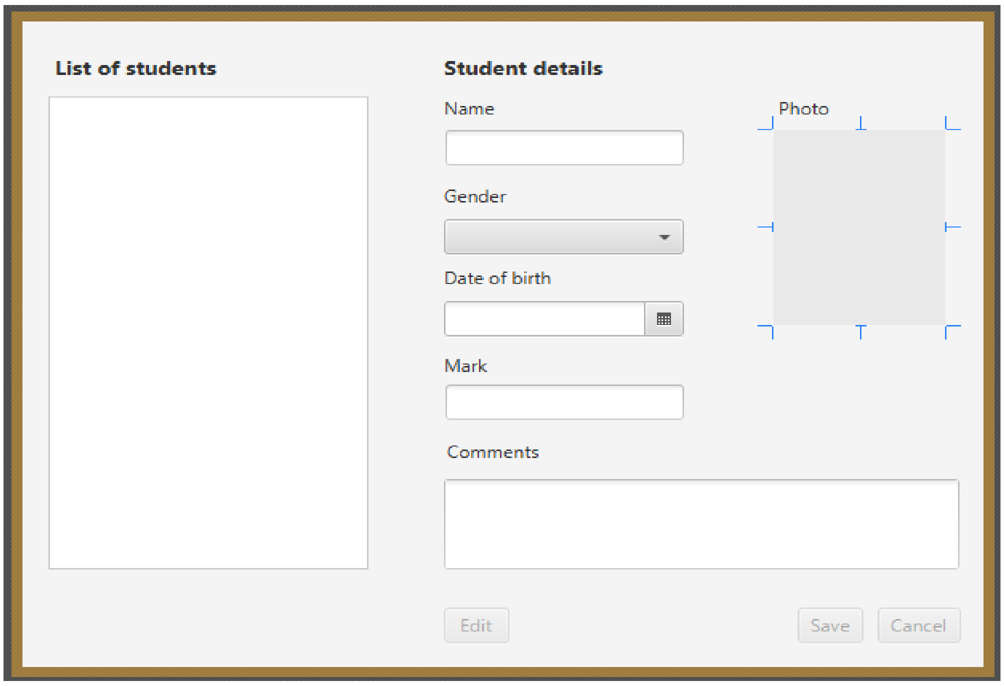
```
public Student fetchStudentByName(String name) {
        Student s = null;
        Connection myConn= Connector();
        try {
                Statement myStmt= myConn.createStatement();
                String sql = "select * from studenttable where name=\""+ name+"\"";
                ResultSet myRs= myStmt.executeQuery(sql);
                while (myRs.next()) {
                        int id=myRs.getInt("id");
                        String gender= myRs.getString("gender");
                        LocalDate birth=null;
                        if (myRs.getDate("dateofbirth")!=null) {
                         birth = myRs.getDate("dateofbirth").toLocalDate();
                        }
                        String photo = myRs.getString("photo");
                        float mark = myRs.getFloat("mark");
                        String comments= myRs.getString("comments");

                        s = new Student(id,name,gender,birth,photo,mark,comments);
                }
                this.close(myConn, myStmt, myRs);
                return s;

        } catch (SQLException e) {
                e.printStackTrace();
                return null;
        }
}
```

TP StudentFX

Step 13 : Update a student

In this section, you have to add the « update student » functionality.

1. Add 3 buttons btnEdit, btnSave and btnCancel to the interface and check their « disable » property in scenebuilder.



2. Once a student is selected, btnEdit is enabled. Add the corresponding code to « displayStudentDetails » method in StudentController.

3. Once btnEdit is pressed, btnSave and btnCancel should be enabled. Write the corresponding method « onEdit » in StudentController and affect it to btnEdit in SceneBuilder throught the « On Action » ComboBox in « Code » Menu.

4. When btnCancel is pressed, btnSave and btnCancel are disabled again. Write the corresponding method « onCancel » in StudentController and affect it to btnCancel in SceneBuilder.

5. To put/update a student's photo, we have to click on the ImageView control. The ImageView works as a file chooser. Write the corresponding method « chooseImage » in StudentController and affect it to the « onMouseClicked » event of the ImageView in SceneBuilder.

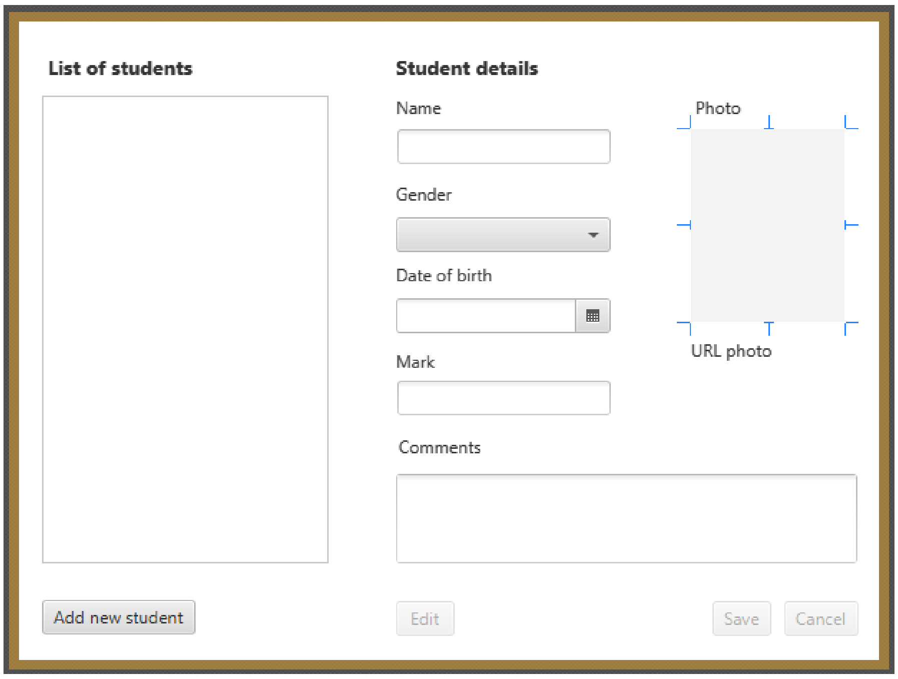
Hint1 : To save the image as a URL, you can add a label « lblURL » to the interface and make it unvisible. You can put the path of the choosed photo in it. This label can be used later to save the photo.
Hint2 (Optional) : In « fetchstudentbyname » and if the student does not have a photo in the database yet (photo = null), you can display a photo of an anonymous student just to fill the ImageView.

6. When btnSave is pressed, the student's details must be updated. Write the corresponding « onSave » method in StudentController. This method should :
   - Take the selected student via « fetchstudentbyname » method in DBManager and save it to a student object.
   - Set its attributes to the values displayed in the controls.
   - Call a new method « updateStudent » to create in DBManager in order to update the student's data in the database.
   - Call the « onCancel » method to disable again btnSave and btnCancel.
7. Write the « updateStudent » method in DBManager.

N.B To make simple, we will not allow modifications on the name (txtName) so that we can use « fetchStudentByName » method to identify the student to update. If we allow modifications on student's name, we should create a method « fetchStudentbyId » to identify the student (using his id) and update it.

Step 14 : Add a new Student

In this section, you have to add the « Create student » functionality.
Add a new button « btnAdd » to the interface.

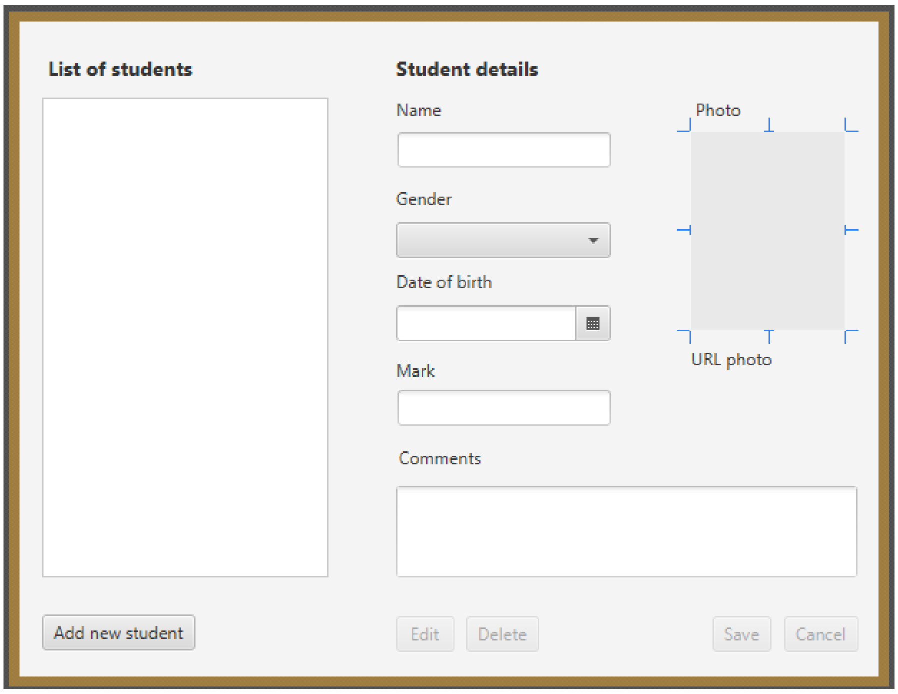


1. Once btnAdd is pressed, btnSave and btnCancel should be enabled and all the controls should be set to blank. The photo can be set to the anonymous. Write the corresponding method « onNew» in StudentController and affect it to btnAdd in SceneBuilder.

2. After filling Student details, we can cancel and get back to the list or save using btnSave. You can use a boolean to distinguish between edit and add. Update the code of « onSave » method in StudentController to support the create functionality:
   - If we are in add mode, declare a new student with no parameter (so you have to add a no-parameter constructor to Student class).
   - Set its attributes to the values displayed in the controls.
   - Call a new method « addStudent » to create in DBManager in order to add the student with his details to the database.
   - Call the « onCancel » method to disable again btnSave and btnCancel.
   - Write the « addStudent » method in DBManager.

Step 15: Delete a student

In this section, you have to add the « delete student » functionality.
Add a new button « btnDelete » to the interface and disable it.



1. Once a student is selected, btnDelete is enabled. Add the corresponding code to « displayStudentDetails » method in StudentController.

2. Once btnDelete is pressed, The selected student is deleted. Write a new « onDelete » method in StudentController and affect it to btnDelete in scenebuilder. This method should :
   - Take the selected student via « fetchstudentbyname » method in DBManager and save it to a student object.
   - Call a new method « deleteStudent » to create in DBManager in order to delete the student's data from the database.
   - Go back to the list of students.
   - (Optional) : you can use a dialog box to ask user before deleting.

Step 15: Optimisation

Optimize the code of your application.