<WebStudentBook>

First Dynamic Web Application
with JSP, Servlets and JDBC

Author : Nada Nahle

In this tutorial, you will learn to develop a simple dynamic web project in Eclipse using MVC design pattern.

The application should allow us to :

1. Display students from a MySQL student database in a browser.
2. Add a new student to the database.
3. Edit a displayed student.
4. Delete a displayed student.

To make simple, a student is charactarized by his :

1. ID.
2. First name
3. Last name
4. Email.

Step 1 : Create the database

Open MySQL Workbench. Create a new database « studentdb » with one table « student ».
The « student » table contains four columns for id (primary key, not null, auto-increment), first_name, last_name and email (Strings).

Step 2 : Tomcat

Make sure that you installed Tomcat and that you connected it to eclipse.

Step 3: Create the application

Open Eclipse IDE and create a new Dynamic Web Project « WebStudentBook ». Precise the target runtime to the Tomcat server that you installed and connected to eclipse. Leave all other defaults and check the « generate web.xml deployment descriptor » before pressing « finish ». You should see your application folder in the project explorer of eclipse.

Step 4: Explore the project

You can see that the project contains a « Java Resources » folder and « WebContent » folder.
In « Java Resources », we can put all java classes (servlets, javabeans and others) later.

Create a new package under « src » default package folder (com.yourName.web.jdbc for example).

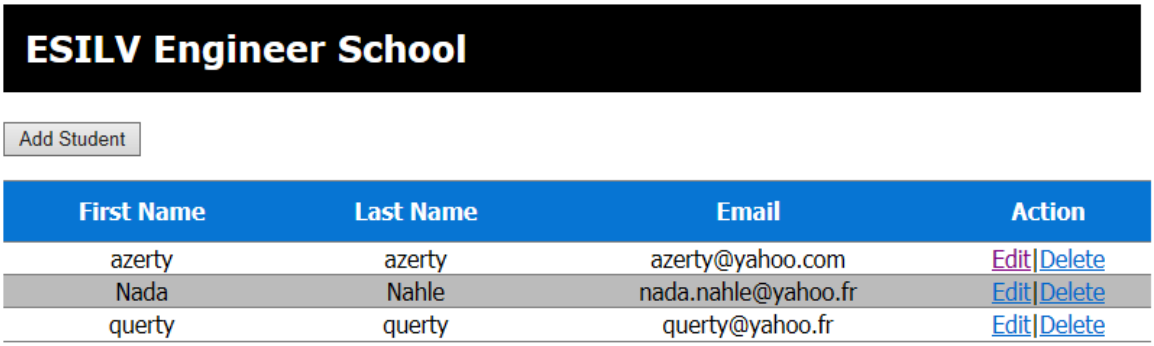
In « WebContent » folder, you can see « META-INF » and « WEB-INF » sub-folders.
In « WEB-INF », we have a « lib » folder. That's where we will put all needed additional libraries (jar files).
In « WebContent », we will put all our view files (JSP, HTML) later.

Step 5: Add MySQL jdbc jar file to the libraries

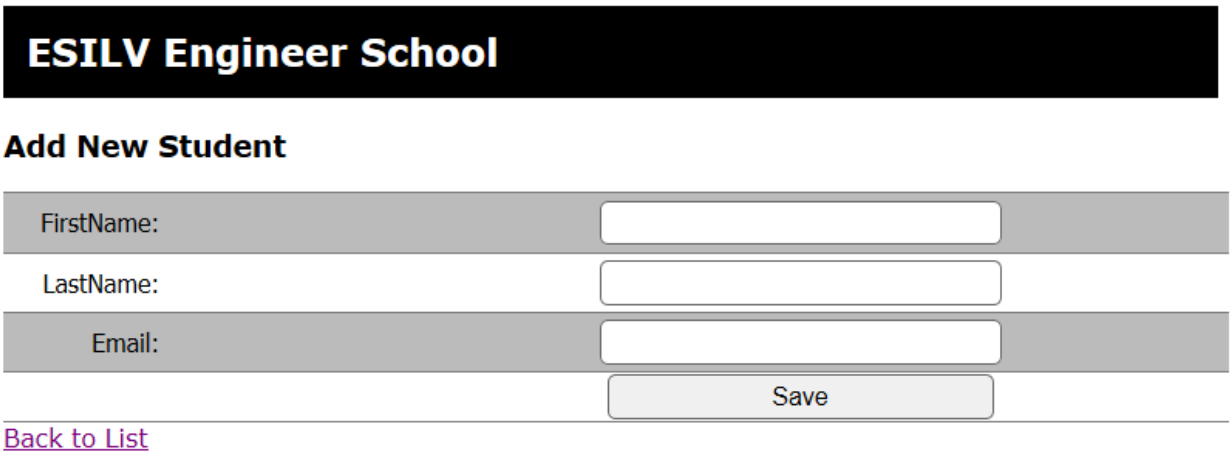
To work with MySQL Database, we have to download « mysql-connector-java » executable jar file and to add it to the « lib » folder under WebContent/WEB-INF in package explorer (eclipse).
You can find this file on BrightSpace in Session 3/ jars.

The final version of our application should be like that :

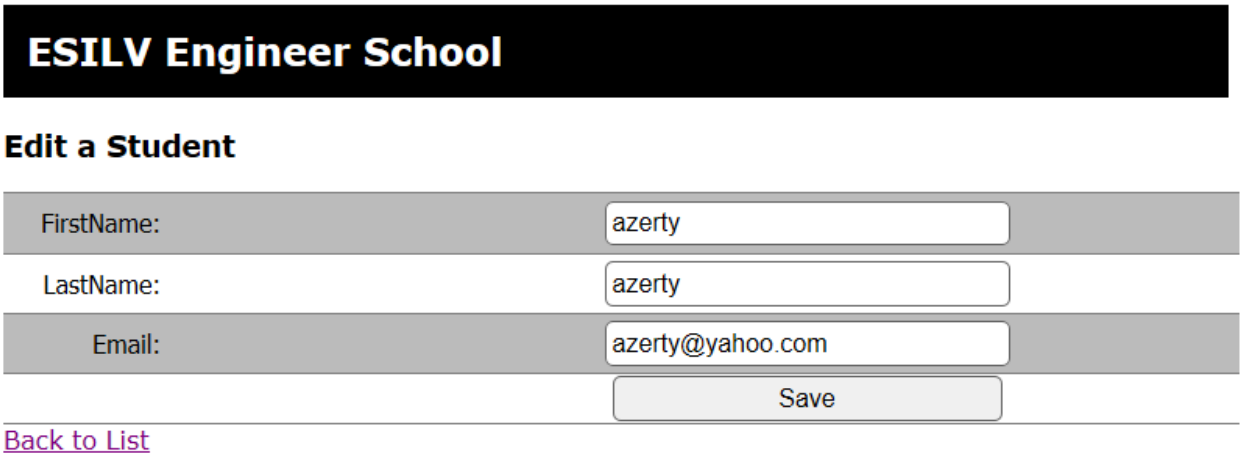
The first page :



If we press the « Add Student » button :



If we press the Edit link for « azerty » student for example:



Step 6: Define a connection pool

Now, once we have the database created and mysql-connector-java in the « lib » folder, we have to create the connection between our application and the database.

In web applications, if we make a single database connection, this will not scale for multiple web users. So, we use « database connection pools » which allows our application to scale and to handle multiple users quickly.

We define a connection pool in META-INF/context.xml. Context.xml is a Tomcat specific file that simply tell tomcat how to connect to the database and how to configure the pool. Create the file and put the following code inside :

```
<Context>

	<Resource name="jdbc/studentdb"
		auth="Container" type="javax.sql.DataSource"
		driverClassName="com.mysql.jdbc.Driver"
		maxActive="20" maxIdle="5" maxWait="10000"
		username="yourMySQLusername" password="yourPassword"
		url="jdbc:mysql://localhost:3306/studentdb?useSSL=false"/>

</Context>
```

We have a ressource reference in the middle. The ressource has :

1. A « name » this name is like alias. We use it to reference the ressource.
2. « Auth » tells Tomcat how we are going to authenticate. « Container » means that it is up to Tomcat server to handle the authentication (the other possible value is « Application »).
3. The « type » is for the java interface that we use to communicate with the pool.
4. Size configurations : maxActive = 20 means thats we have 20 connections in the pool. MaxIdle=5 means thats when we have no users in our system, we have maximum 5 connections available. « MaxWait » is to set in milliseconds the maximal waiting time before getting a connection from the pool.
5. « username », « password » and « driverClassName » tell Tomcat how to connect to the database.
6. « url » sets the url for the database.

Step 7: Get the connection pool in java code

To get the connection pool in java code, Java EE has a technique called « ressource injection » for servlets. This means that Tomcat will automagically set the connection pool/ datasource on the servlet by using ressource annotation (@Resource).
To test our connection pool, create a new serlet « TestServlet » in the package that you created in step 4. Uncheck « contructors from superclass » and « doPost » while creating. You will have a default servlet sample created by eclipse. You can add the ressource injection for the connection pool and declare a Datasource attribute to handle it.

```
@WebServlet("/TestServlet")
public class TestServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;

	@Resource(name="jdbc/studentdb")
	private DataSource dataSource;
```

Fix the imports for Resource and DataSource. Be carefull to add the import of javax.sql for the datasource.

The ressource name is exactly the same name used in context.xml file.

In doGet Method, we will add some code to test our connection.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
		// TODO Auto-generated method stub

		//Step1: set up the printwriter
		PrintWriter out = response.getWriter();
		response.setContentType("text/plain");

		//Step2: Get a connection to the database
		Connection myConn=null;
		Statement myStmt= null;
		ResultSet myRs=null;

		try{
			myConn= dataSource.getConnection();
			//Step3: create SQL statements
			String sql = "select * from student";
			myStmt= myConn.createStatement();
			//Step4: Execute SQL query
			myRs=myStmt.executeQuery(sql);
			//Step5: Process the ResultSet
			while(myRs.next()){
				String email = myRs.getString("email");
				out.println(email);
			}
		}catch(Exception exc){
			System.out.println(exc.getMessage());
		}
	}
```

Fix the imports of Connection, Statement, ResultSet respectively with java.sql.Connection and java.sql.Statement and java.sql.ResultSet.

Now, you can test the connection by running the « TestServlet » on the server.
Right-click on the servlet in the project explorer and click on « Run As » and select « Run on Server ». Choose the Tomcat server that you installed and connected to eclipse. Click on next and finish. You should see the students' emails on the localhost browser.

Step 8: List student functionality

Let's start by the first functionlity that our application should satisfy : Listing the students.

Since we use MVC design pattern for our application, we need to create :

- The Model
- The Controller
- The View

In Web applications, a model is constituted by JavaBeans and other Helper classes. A controller is a servlet and a view is a JSP file.

- Create a new Servlet « StudentControllerServlet » under you java package. To make simple,

uncheck « contructors from superclass » and « doPost » while creating.
- Create a new JSP file « list-students » under WebContent.

For our model, we first need to create a student javabean. Then, and in order too communicate with the database, we will use the DAO (Data Accessor Object) well-known design pattern.
In DAO, the controller doesn't communicate directly with the database but through a helper class. This helper class is responsible for interfacing with the database.

- Create a new class « Student » under your java package.
- Create a new class « StudentDBUtil » under your java package. It will be our helper class.

So, we prepared our application to respect the MVC architecture. Let's move to coding now.

Step 9: Write the JavaBean code

We will start with the « Student » class. Define four private attributes for id, first_Name, last_Name and email. Use eclipse to generate getters and setters, two constructors (1 with all the fields and 1 without the id field since it can be auto-incremented in the database) and a ToString() method with all fields.

Step 10: Start the Controller code

Before writing the code for StudentDBUtil, let us simulate a little bit the behavior of the application.

As you know, the servlet is launched once the client enters a url in the browser. So it is up to the servlet to set the connection pool before calling the StudentDBUtil, which is responsible for interfacing with the « studentdb » database. Therefore, the reference to the Resource that we created in « context.xml » should be declared in the StudentControllerServlet and handled by a DataSource private field. Since the StudentControllerServlet needs to call the StudentDBUtil, we need also a private StudentDBUtil field.

Open « StudentControllerServlet » and add the following code :

```
@WebServlet("/StudentControllerServlet")
public class StudentControllerServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;

	private StudentDbUtil studentDbUtil;

	@Resource(name="jdbc/studentdb")
	private DataSource dataSource;

	protected void doGet(HttpServletRequest request, HttpServletResponse response)
	throws ServletException, IOException {
	}

}
```

You can see that « StudentControllerServlet » extends httpservlet and has the annotation `@WebServlet("/StudentControllerServlet")` which means that the « StudentControllerServlet » is called when the url « http://localhost:8080/WebStudentBook/StudentControllerServlet » is entered in the browser.

The connection pool should be initialized once the servlet is initialized. You can generate the

« init » method using eclipse by doing right-click on the code page → source → override/implement methods → generic servlet → init.

```
@Override
	public void init() throws ServletException {
		// TODO Auto-generated method stub
		super.init();
		studentDbUtil = new StudentDbUtil(dataSource);
	}
```

The « doGet » method is empty for now, we will fill it later. But now let us care about StudentDBUtil helper class.

Step 11: Write the StudentDBUtil class code

In StudentDBUtil class, we will manage the communication between the « StudentControllerServlet » the « studentdb » database.

You see that in Step 10, we needed to create a studentDBUtil object with a datasource parameter in order to communicate the connection pool initialized in « StudentControllerServlet » to the « StudentDBUtil ». The « StudentDBUtil » should have one private field for the datasource. The constructor should take a datasource parameter to handle the datasource connection pool sent by the servlet.

```
public class StudentDbUtil {

	private DataSource dataSource;

	public StudentDbUtil(DataSource theDataSource) {
		dataSource = theDataSource;
	}
```

To display the student list, we need a method to fetch the students from the database in StudentDBUtil. This method does not require any parameter and should return a list of students. The code is similar to what we did in TestServlet : we get a connection from the pool, we create a statement, we execute the statement and put the reslut in a ResultSet. We explore the ResultSet and we create a new Student object for each row in the student table of the database. Finally we add the Student object to the student list. The final list is returned back by the method to the caller.

```
public List<Student> getStudents() throws Exception {
		List<Student> students= new ArrayList<Student>();

		Connection myConn=null;
		Statement myStmt = null;
		ResultSet myRs= null;

		try {
			myConn = dataSource.getConnection();
			myStmt= myConn.createStatement();
			String sql= "select * from student order by last_name";
			myRs = myStmt.executeQuery(sql);

			while(myRs.next()){
				int id = myRs.getInt("id");
				String firstName=myRs.getString("first_name");
				String lastName=myRs.getString("last_name");
				String email = myRs.getString("email");
```

```
                Student tempStudent= new Student(id,firstName,lastName,email);
                        students.add(tempStudent);
                }

                return students;

        } finally{
                close(myConn,myStmt,myRs);

        }
    }

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 myConn.close does not close the connection. It just returns back the connection to the connection pool.

Step 12: Complete the controller

In this step, let us move back to the StudentControllerServlet and write the « doGet » method.
The doGet Method is called when the servlet is launched. Here, the principal role of « StudentControllerServlet » is to list the students of the database. It should therefore call the « getStudents » method of StudentDBUtil to get the returned list of students. It should then call the JSP page to display the students in the browser.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

        try {
                listStudents(request,response);
        } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

    }

private void listStudents(HttpServletRequest request, HttpServletResponse response)
throws Exception{

List<Student> students = studentDbUtil.getStudents();
request.setAttribute("STUDENT_LIST", students);
RequestDispatcher dispatcher = request.getRequestDispatcher("/list-students.jsp");
dispatcher.forward(request, response);
}
```

You can see that we send the list of students « STUDENT-LIST » to the view through a « setAttribute » method of request. Finally, the dispatcher calls the « list-students » jsp file.

Step 13: Write the JSP file without JSLT (with scriplets) and without styling

Open the « list-students.jsp » under « WebContent » and write this code inside :

```
<%@ page import="java.util.*,com.nada.web.jdbc.*" %>

<html>
<head>
<title>Web Student Book</title>

</head>
<% List<Student> theStudents = (List<Student>)request.getAttribute("STUDENT_LIST"); %>
<body>
<!--  ${STUDENT_LIST}-->
<%= theStudents %>
</body>
</html>
```

Note that we need to import two packages « java.util.* » and « com.nada.web.jdbc.* » because we use « List » collection and « Student » class. A jsp file can contain java code (scriplets)
<% java code %>
Note also that we have an access to the attribute sent by the servlet through « request.getattribute ».
Finally, we can display the students without any styling through the
« jsp expression **<%= the students %>** »

Run your « StudentControllerServlet » and look at the result in the browser. You should have the students as string concatenation like you defined in toString() method of « Student » class.

Step 14: Display the students in a table and style the page with CSS

Let us display the students in a table and add a .css file to style the page. For that, create a folder (css) under WebContent. Copy the file « style.css » from BrightSpace Session3/style.css and connect it to the list-students.jsp with the link tag.

```
<%@ page import="java.util.*,com.nada.web.jdbc.*" %>

<html>
<head>
<title>Web Student Tracker</title>
<link type="text/css" rel="stylesheet" href="css/style.css">
</head>
<% List<Student> theStudents = (List<Student>)request.getAttribute("STUDENT_LIST"); %>
<body>
<%=theStudents %>
<div id="wrapper">
	<div id="header">
		<h2>ESILV Engineer School</h2>
	</div>
</div>
<div id="container">
	<div id="content">
	<table>
		<tr>
			<th>First Name </th>
			<th>Last Name</th>
			<th>Email </th>
		</tr>
		<% for(Student tempStudent:theStudents) { %>
			<tr>
```

```
                        <td><%= tempStudent.getFirstName() %></td>
                        <td><%= tempStudent.getLastName() %></td>
                        <td><%= tempStudent.getEmail() %></td>
                </tr>
                <%} %>
        </table>
</div>
</div>

</body>
</html>
```

You should notice that scriplets make the jsp page hard to read. Therefore, we will replace scriplets by JSTL.

Step 15: Write the JSP page with JSLT

With JSLT, we have no more java code in our « list-students.jsp ». That's better and much more visible. But to use JSLT, we should include the taglib in the head of the file and add 2 jars for JSLT in our « lib » folder in « WEB-INF ». You can find the jars on BrightSpace Session3/jars.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>
<head>
<title>Web Student Tracker</title>
<link type="text/css" rel="stylesheet" href="css/style.css">
</head>
<body>
<!--  ${STUDENT_LIST}-->
<div id="wrapper">
        <div id="header">
                <h2>ESILV Engineer School</h2>
        </div>
</div>
<div id="container">
        <div id="content">
        <table>
                <tr>
                        <th>First Name </th>
                        <th>Last Name</th>
                        <th>Email </th>
                </tr>
                <c:forEach var="tempStudent" items="${STUDENT_LIST }" >
                        <tr>
                                <td> ${tempStudent.firstName}</td>
                                <td> ${tempStudent.lastName}</td>
                                <td> ${tempStudent.email}</td>
                </c:forEach>
        </table>
</div>
</div>
</body>
</html>
```

Run the servlet and look to the Result. Much better no ? !

Step 16: Adding a welcome file

Currently, to access our MVC application, we need to explicitly call the servlet URL.
http://localhost:8080/WebStudentBook/StudentControllerServlet
But what we need, is to just access our application without explicitly calling the servlet.
Java Servlet spec defines a deployment descriptor file : WEB-INF/web.xml. This file is processed in

order in case we don't have an explicit call for the servlet in the URL. We can add a welcome file line for our servlet in the top of the list.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>web-student-tracker</display-name>
  <welcome-file-list>
      <welcome-file>StudentControllerServlet</welcome-file>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Now you can test the result by opening a browser and testing the following url : http://localhost:8080/WebStudentBook

It should work and you should have all the students well displayed in a table.

Add a Student Functionality :

To add a student, we need a form that allows us to enter the student's information (fisrt name, last name and email). Then, if we press the button « save » the student is saved to the database and we get back to the list of students including the new one. If we change our mind and we don't want to save a new student anymore, we can use a link to the list of students without saving a new student (like a cancel button).

To start this functionality, add a button « Add new Student » on the top of « list-students.jsp ».

```
<div id="content">
	<form action="AddStudentServlet" method="get">
	<input type="submit" value="Add Student"/>
	</form>
	<table>
	…
```

The action attribute of the form indicates which servlet has to be called when we press the button.
The method attribute indicates wich method is to be called in the servlet.

I choose to manage the « Add Student Functionality » in a new servlet.
Create a new Servlet « AddStudentServlet » with a « doGet » method.
In the « doGet » method, we only want to open a new JSP to enter a new student's information.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
		// TODO Auto-generated method stub
	request.getRequestDispatcher("/add-student.jsp").forward(request, response);
	}
```

Create a new JSP « add-student.jsp » and write this code inside :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<link type="text/css" rel="stylesheet" href="css/add-student-style.css">
<link type="text/css" rel="stylesheet" href="css/style.css">

<title>Add a Student</title>
</head>
<body>
<div id="wrapper">
	<div id="header">
		<h2>ESILV Engineer School</h2>
	</div>
</div>
<div id="container">
	<h3> Add New Student</h3>

<form action="StudentControllerServlet" method = "post">
	<table>
		<tbody>
			<tr>
```

```
                        <td><label>FirstName: </label> </td>
                        <td><input type="text" name = "firstName"/></td>
                    </tr>
                    <tr>
                        <td><label>LastName: </label> </td>
                        <td><input type="text" name = "lastName"/></td>
                    </tr>
                    <tr>
                        <td><label>Email: </label> </td>
                        <td><input type="text" name = "email"/></td>
                    </tr>
                    <tr>
                        <td><label></label> </td>
                        <td><input type="submit" value = "Save"/></td>
                    </tr>
                </tbody>
        </table>

</form>
<div style="clear:both;"></div>
        <a href="StudentControllerServlet">Back to List</a>
</div>

</body>
</html>
```

I added a style for the new JSP. You can find the « add-student-style.css » on BrightSpace. Add it to the « css folder » under WebContent. You can see that in the JSP page, we create a form that should contain the student's information (first name, last name and email). We have also a submit button « Save ».
The action of the form is set to the « StudentControllerServlet » and the method is « doPost » which means that the information of the student will be sent as parameters to the servlet and we will be able to access them in the « doPost » method of the servlet.

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
            String fn= req.getParameter("firstName");
            String ln= req.getParameter("lastName");
            String email = req.getParameter("email");
            Student student = new Student(fn,ln,email);
            studentDbUtil.addStudent(student);
            try {
                listStudents(req,resp);
            } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

}
```

We can see that we left the « Add student » functionality to the studentDbUtil object since it is the interface between our servlet and the database. So, we have to add the « addStudent » method to the StudentDBUtil class. Once, the saving process is completed, we can call the « listStudents » method of the same servlet « studentControllerServlet » to show the new list of students.

Add the « addStudent » method to the StudentDBUtil class.

```
public void addStudent(Student student){
        Connection myConn=null;
        PreparedStatement myStmt = null;
```

```
        ResultSet myRs= null;

        try {
                myConn = dataSource.getConnection();
                String sql = "INSERT INTO Student (first_name, last_name, email)
                              VALUES (?, ?, ?)";
                myStmt = myConn.prepareStatement(sql);
                    String firstName = student.getFirstName();
                    String lastName = student.getLastName();
                    String email = student.getEmail();

                myStmt.setString(1, firstName);
                myStmt.setString(2, lastName);
                myStmt.setString(3, email);
                myStmt.execute();
        }
        catch(Exception e){
                System.out.println(e.getMessage());
        }

        finally{
                close(myConn,myStmt,myRs);
        }

}
```

Run the application and test the new functionality.

Edit a Student Functionality :

To edit a student, we first need to add a button Edit for each row of the students' list. We can use a link button. Once this button is clicked, a new JSP « edit-student » is called. « edit-student » is similar to « add-student » but prefilled with the selected student's information. We can modify this information and save the student or we can go back to the list without saving.

To add the button link to the table of students, we use the following code in « list-students.jsp ».

```
<c:forEach var="tempStudent" items="${STUDENT_LIST }" >

            <c:url var="EditLink" value= "EditStudentServlet">
            <c:param name="studentId" value="${tempStudent.id}"/>
            </c:url>
                  <tr>
                        <td> ${tempStudent.firstName}</td>
                        <td> ${tempStudent.lastName}</td>
                        <td> ${tempStudent.email}</td>
                        <td> <a href="${EditLink }"> Edit</a></td>
                  </tr>
            </c:forEach>
```

We use here a new feature of JSTL : the url tag
url tag can call a servlet and carry some parameters with the request. Here, we need to carry the student id. Then, in each row we add a column for the button link that points to url of the servlet.

Create a new servlet « EditStudentServlet » with init, doGet and doPost methods.
When the servlet is called by the link button « Edit » of « list-students.jsp « , the « doGet » method

is called.
The « doGet » method retrieves the « id » parameter sent with the url link and use an instance of the « StudentDBUtil » class to fetch the student with the given « id » from the database. This student is set as request attribute and the new JSP « edit-student.jsp » is called to show the edit form of this specific student.

```
@WebServlet("/EditStudentServlet")
public class EditStudentServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;

	private StudentDbUtil studentDbUtil;

	@Resource(name="jdbc/studentdb")
	private DataSource dataSource;

	int id;

	@Override
	public void init() throws ServletException {
		super.init();
		studentDbUtil = new StudentDbUtil(dataSource);
	}

    public EditStudentServlet() {
        super();
        // TODO Auto-generated constructor stub
    }
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
	id=Integer.parseInt(request.getParameter("studentId"));
	Student student= studentDbUtil.fetchStudent(id);
	request.setAttribute("Student", student);
	request.getRequestDispatcher("edit-student.jsp").forward(request, response);
	}
```

And in StudentDBUtil, we add the « fetchStudent » method :

```
public Student fetchStudent(int id) {
	Connection myConn=null;
	Statement myStmt = null;
	ResultSet myRs= null;
	Student student=null;

	try {
		myConn = dataSource.getConnection();
		myStmt= myConn.createStatement();
		String sql= "select * from student where id="+id;
		myRs = myStmt.executeQuery(sql);

		while(myRs.next()){
			String firstName=myRs.getString("first_name");
			String lastName=myRs.getString("last_name");
			String email = myRs.getString("email");

			student = new Student(id,firstName,lastName,email);
		}

		return student;
	}catch(Exception e){
		System.out.println(e.getMessage());
```

```
            return null;
      } finally{
            close(myConn,myStmt,myRs);

      }
}
```

The « edit-student.jsp » takes care of the presentation of the edit form just like the « add-student.jsp ».

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<link type="text/css" rel="stylesheet" href="css/add-student-style.css">
<link type="text/css" rel="stylesheet" href="css/style.css">

<title>Edit a student</title>
</head>
<body>
<div id="wrapper">
      <div id="header">
            <h2>ESILV Engineer School</h2>
      </div>
</div>
<div id="container">
      <h3> Edit a Student</h3>
<form action="EditStudentServlet" method = "post">
<table>
      <tbody>
      <tr>
      <td><label>FirstName: </label> </td>
      <td><input type="text" name = "firstName" value="$ Student.firstName }"/></td>
      </tr>
      <tr>
      <td><label>LastName: </label> </td>
      <td><input type="text" name = "lastName" value="${Student.lastName }"/></td>
      </tr>
      <tr>
      <td><label>Email: </label> </td>
      <td><input type="text" name = "email" value="${Student.email }"/></td>
      </tr>
      <tr>
      <td><label></label> </td>
      <td><input type="submit" value = "Save"/></td>
      </tr>
      </tbody>
</table>
</form>
<div style="clear:both;"></div>
      <a href="StudentControllerServlet">Back to List</a>
</div>
</body>
</html>
```

Here, we can see that this form has the « EditStudentServlet » as action and calls the « doPost » method. When the « Save » button is clicked, the « doPost » method of the « EditStudentServlet » is called and the student's information are carried as paramaters on the request.

We add the code of « doPost » method as following. Once we retrieve the paramaters, we can create a new Student object. Then, we can use the StudentDBUtil to update the student. Finally, when the student is updated, we redirect the request to the initial servlet « StudentControllerServlet » to display the updated list of students again.

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

	String fn= request.getParameter("firstName");
	String ln= request.getParameter("lastName");
	String email = request.getParameter("email");
	Student student = new Student(id,fn,ln,email);
	studentDbUtil.updateStudent(student);
	response.sendRedirect("StudentControllerServlet");
}
```

As we can see, we still have to write the studentDBUtil. updateStudent(student). So, in the StudentDBUtil class, add the update method :

```
public void updateStudent(Student student) {
	Connection myConn=null;
	PreparedStatement myStmt = null;

	try {
		myConn = dataSource.getConnection();
	String sql = "update student set first_name=?, last_name=?,email=? where id=?";
		myStmt = myConn.prepareStatement(sql);

		myStmt.setString(1, student.getFirstName());
		myStmt.setString(2, student.getLastName());
		myStmt.setString(3, student.getEmail());
		myStmt.setInt(4,student.getId());
		myStmt.execute();
	}
	catch(Exception e){
		System.out.println(e.getMessage());
	}

	finally{
		close(myConn,myStmt,null);
	}
}
```

Run the application and test the new functionality.

Delete a Student Functionality :

To delete a student, we first need to add a button « Delete » besides the «Edit » one at each row of the students' list. We can also use a link button. Once this button is clicked, a new servlet « DeleteStudentServlet » is called (the doGet method of the servlet). The link should carry the

selected student's id. In the « doGet » method, we can retrieve the student's id. Then, we can make use of StudentDBUtil to delete the student having this « id » and we redirect the request to the StudentControllerServlet to show the list after deleting the student.

In « list-students.jsp :

```
<c:forEach var="tempStudent" items="${STUDENT_LIST }" >
	<c:url var="EditLink" value= "EditStudentServlet">
	<c:param name="studentId" value="${tempStudent.id}"/>
	</c:url>
	<c:url var="DeleteLink" value= "DeleteStudentServlet">
	<c:param name="studentId" value="${tempStudent.id}"/>
	</c:url>
	<tr>
	<td> ${tempStudent.firstName}</td>
	<td> ${tempStudent.lastName}</td>
	<td> ${tempStudent.email}</td>
	<td> <a href="${EditLink }"> Edit</a>|<a href="${DeleteLink }">Delete</a></td>
	</tr>
</c:forEach>
```

In the Servlet :

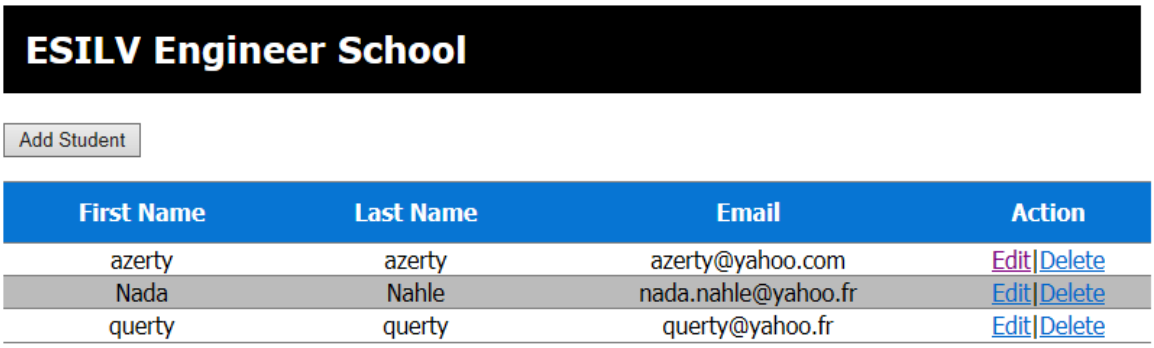
```
@WebServlet("/DeleteStudentServlet")
public class DeleteStudentServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;
	private StudentDbUtil studentDbUtil;

	@Resource(name="jdbc/studentdb")
	private DataSource dataSource;

	@Override
	public void init() throws ServletException {
		super.init();
		studentDbUtil = new StudentDbUtil(dataSource);
	}

	protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

		int id=Integer.parseInt(request.getParameter("studentId"));
		studentDbUtil.deleteStudent(id);
		response.sendRedirect("StudentControllerServlet");
	}
}
```

In StudentDBUtil :

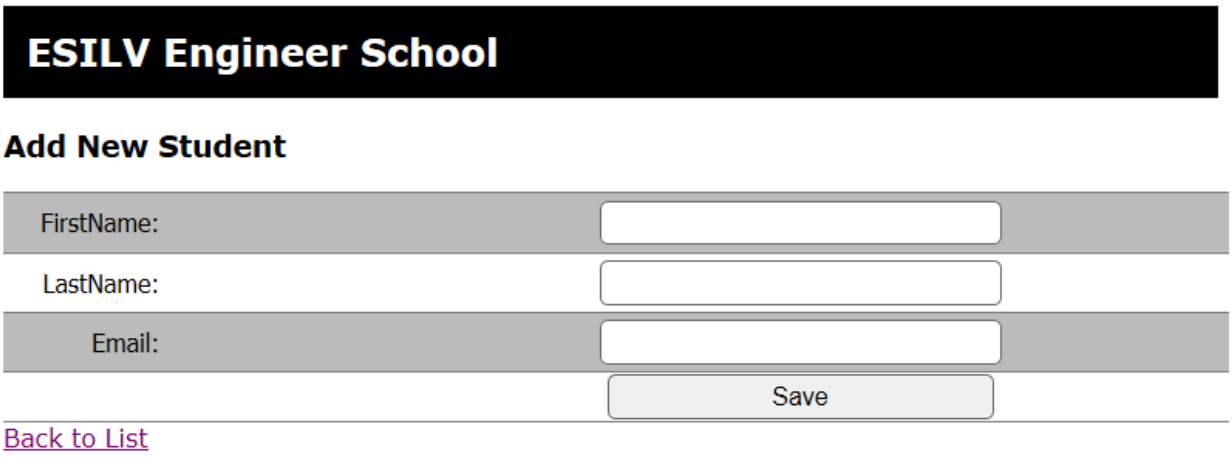
```
public void deleteStudent(int id) {
		// TODO Auto-generated method stub
		Connection myConn=null;
		Statement myStmt = null;
		try {
			myConn = dataSource.getConnection();
			myStmt= myConn.createStatement();
			String sql= "delete from student where id="+id;
			myStmt.execute(sql);
			}catch(Exception e){
			System.out.println(e.getMessage());
		} finally{ close(myConn,myStmt,null);  }}
```

The final version of our application should be like that :

The first page :



If we press the « Add Student » button :



If we press the Edit link for « azerty » student for example: