

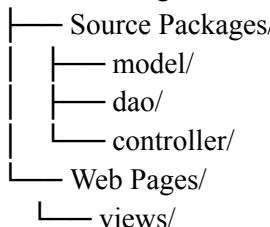
EXERCISE 1: PROJECT SETUP & MODEL LAYER (15 points)

Estimated Time: 25 minutes

Task 1.1: Create Project Structure (5 points)

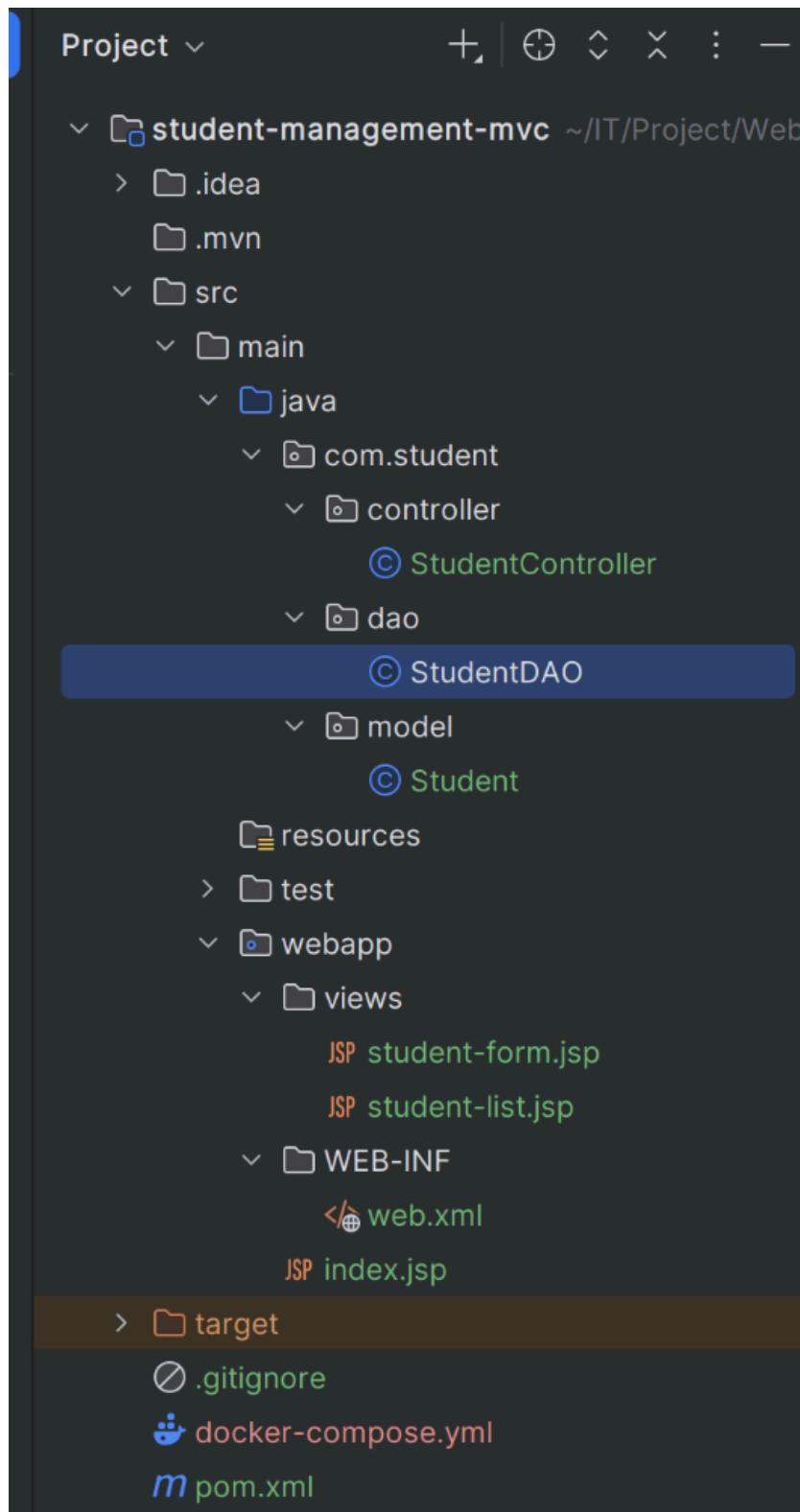
Create proper MVC folder structure:

StudentManagementMVC/



Steps:

1. Create new Web Application project: **StudentManagementMVC**
2. Create Java packages: **model**, **dao**, **controller**
3. Create **views** folder inside Web Pages
4. Add MySQL Connector/J library



Task 1.2: Create Student JavaBean (5 points)

File: `src/model/Student.java`

Requirements:

- Private attributes: id, studentCode, fullName, email, major, createdAt
- Public no-arg constructor
- Public parameterized constructor (without id)
- Public getters and setters for all attributes
- Override `toString()` method

```
public class Student {
    private int id;
    private String studentCode;
    private String fullName;
    private String email;
    private String major;
    private Timestamp createdAt;

    // No-arg constructor (required for JavaBean)
    public Student() {
    }

    // Constructor for creating new student (without ID)
    public Student(String studentCode, String fullName, String email, String major) {
        this.studentCode = studentCode;
        this.fullName = fullName;
        this.email = email;
        this.major = major;
    }

    // Getters and Setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getStudentCode() {
        return studentCode;
    }

    public void setStudentCode(String studentCode) {
        this.studentCode = studentCode;
    }

    public String getFullName() {
        return fullName;
    }

    public void setFullName(String fullName) {
        this.fullName = fullName;
    }

    public String getEmail() {
```

```

        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getMajor() {
        return major;
    }

    public void setMajor(String major) {
        this.major = major;
    }

    public Timestamp getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(Timestamp createdAt) {
        this.createdAt = createdAt;
    }

    @Override
    public String toString() {
        return "Student{" +
            "id=" + id +
            ", studentCode='" + studentCode + '\'' +
            ", fullName='" + fullName + '\'' +
            ", email='" + email + '\'' +
            ", major='" + major + '\'' +
            '}';
    }
}

```

1. Private attributes

All core data fields are declared private to ensure robust encapsulation:

- + private int id;
- + private String studentCode;
- + private String fullName;
- + private String email;
- + private String major;
- + private Timestamp createdAt;

2. Public no-arg constructor

A default, public no-argument constructor is included, fulfilling the JavaBean requirement for easy object instantiation by frameworks:

```
public Student() {  
}  
}
```

3. Public parameterized constructor (without id)

A specific public constructor is provided for creating new records, excluding the auto-generated id and createdAt:

```
public Student(String studentCode, String fullName, String email, String major) {  
  
    this.studentCode = studentCode;  
  
    this.fullName = fullName;  
  
    this.email = email;  
  
    this.major = major;  
  
}
```

4. Public getters and setters for all attributes

Standard public accessor methods are implemented for all fields, ensuring data can be read and written externally (e.g., getter for id and setter for fullName):

```
public int getId() {  
  
    return id;  
}  
  
  
public void setId(int id) {  
  
    this.id = id;  
}  
  
  
public String getStudentCode() {  
}
```

```
        return studentCode;
    }

public void setStudentCode(String studentCode) {
    this.studentCode = studentCode;
}

public String getFullName() {
    return fullName;
}

public void setFullName(String fullName) {
    this.fullName = fullName;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getMajor() {
    return major;
}

public void setMajor(String major) {
```

```
this.major = major;  
}  
  
public Timestamp getCreatedAt() {  
    return createdAt;  
}  
  
public void setCreatedAt(Timestamp createdAt) {  
    this.createdAt = createdAt;  
}
```

5. Override `toString()` method

The `toString()` method is overridden to provide a clear string representation of the object's state for debugging and logging purposes:

```
@Override  
public String toString() {  
    return "Student{" +  
        "id=" + id +  
        ", studentCode=\"" + studentCode + '\"' +  
        ", fullName=\"" + fullName + '\"' +  
        ", email=\"" + email + '\"' +  
        ", major=\"" + major + '\"' +  
    '}';  
}
```

Task 1.3: Create StudentDAO (5 points)

File: `src/dao/StudentDAO.java`

Requirements:

- Constants for database configuration
- `getConnection()` method
- `getAllStudents()` method - returns `List<Student>`
- Use try-with-resources
- Proper exception handling

Test Your DAO:

```
// Add this main method to test (remove after testing)
public static void main(String[] args) {
    StudentDAO dao = new StudentDAO();
    List<Student> students = dao.getAllStudents();
    for (Student s : students) {
        System.out.println(s);
    }
}
```

Checkpoint #1: Show instructor that Student model and DAO work correctly.

```
private Connection getConnection() throws SQLException {
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
        return DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
    } catch (ClassNotFoundException e) {
        throw new SQLException("MySQL Driver not found", e);
    }
}

// Get all students
public List<Student> getAllStudents() {
    List<Student> students = new ArrayList<>();
    String sql = "SELECT * FROM students ORDER BY id DESC";

    try (Connection conn = getConnection();
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(sql)) {

        while (rs.next()) {
            Student student = new Student();
            student.setId(rs.getInt("id"));
            student.setName(rs.getString("name"));
            student.setAge(rs.getInt("age"));
            student.setGpa(rs.getDouble("gpa"));
            student.setMajor(rs.getString("major"));
            student.setMinor(rs.getString("minor"));
            student.setHobbies(rs.getString("hobbies"));
            student.setNotes(rs.getString("notes"));
            students.add(student);
        }
    } catch (SQLException e) {
        throw new RuntimeException("Error executing query: " + e.getMessage());
    }
    return students;
}
```

```
        student.setStudentCode(rs.getString("student_code"));
        student.setFullName(rs.getString("full_name"));
        student.setEmail(rs.getString("email"));
        student.setMajor(rs.getString("major"));
        student.setCreatedAt(rs.getTimestamp("created_at"));
        students.add(student);
    }

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

return students;
}

  ↴
  ↵  dao
  ↵  ↴ StudentDAO
  ↵  ↵  model
  ↵  ↵  ↴ Student
  ↵  ↵  resources
  ↵  ↵  test
  ↵  ↵  webapp
  ↵  ↵  ↵  views
  ↵  ↵  ↵  ↴ student-form.jsp
  ↵  ↵  ↵  ↴ student-list.jsp
  ↵  ↵  ↵  WEB-INF
  ↵  ↵  ↵  ↴ web.xml
  ↵  ↵  ↵  ↴ index.jsp
  ↵  ↵ target
  ↵  .gitignore
  ↵  docker-compose.yml

Run  StudentDAO ×
```

The screenshot shows an IDE interface with a code editor, a file tree, and a run terminal.

- Code Editor:** Displays Java code for a DAO class. The code includes catching a SQLException, printing the stack trace, and returning a list of students.
- File Tree:** Shows the project structure:
 - dao: contains StudentDAO
 - model: contains Student
 - resources
 - test
 - webapp:
 - views:
 - student-form.jsp
 - student-list.jsp
 - WEB-INF:
 - web.xml
 - index.jsp
 - target
 - .gitignore
 - docker-compose.yml
- Run Terminal:** Shows the output of running the code. It prints three student objects to the console:

```
/usr/lib/jvm/java/bin/java -javaagent:/home/fuduweili/.local/share/JetBrains/Toolbox/apps/intelliij-idea-ultimate/lib/idea_rt.jar=41 Pe
Student{id=3, studentCode='SV003', fullName='Le Van C', email='c.le@example.com', major='Software Engineering'}
Student{id=2, studentCode='SV002', fullName='Tran Thi B', email='b.tran@example.com', major='Information Technology'}
Student{id=1, studentCode='SV001', fullName='Nguyen Van A', email='a.nguyen@example.com', major='Computer Science'}
```

Process finished with exit code 0

EXERCISE 2: CONTROLLER LAYER (20 points)

Estimated Time: 40 minutes

Task 2.1: Create Basic Servlet (10 points)

File: `src/controller/StudentController.java`

Requirements:

- Extend `HttpServlet`
- Use `@WebServlet("/student")` annotation
- Override `init()` to initialize StudentDAO
- Override `doGet()` to handle GET requests
- Implement `listStudents()` method
- Forward to JSP view

Evaluation Criteria:

Criteria	Points
Servlet properly configured	2
<code>init()</code> initializes DAO	2
<code>doGet()</code> routes correctly	2
<code>listStudents()</code> gets data from DAO	2
Sets request attribute	1
Forwards to JSP	1

Extend HttpServlet and Use @WebServlet("/student")

The servlet is properly configured by extending the base class and using the correct annotation for URL mapping:

```
@WebServlet("/student")
public class StudentController extends HttpServlet {
```

Override init() to initialize StudentDAO

The `init()` method is overridden to ensure the StudentDAO is initialized and ready for use upon servlet loading:

```
@Override  
  
public void init() {  
  
    studentDAO = new StudentDAO();  
  
}
```

Override doGet() to route correctly

The doGet() method correctly handles the default action ("list") when no specific action parameter is provided:

```
@Override  
  
protected void doGet(HttpServletRequest request, HttpServletResponse  
response)  
  
throws ServletException, IOException {  
  
    String action = request.getParameter("action");  
  
    if (action == null) {  
  
        action = "list";  
  
    }  
  
    switch (action) {  
  
        case "new":  
  
            showNewForm(request, response);  
  
            break;  
  
        case "edit":  
  
            showEditForm(request, response);  
  
            break;  
  
        case "delete":  
  
            deleteStudent(request, response);  
    }  
}
```

```
        break;

    default:

        listStudents(request, response);

        break;

    }

}
```

Implement listStudents(): Get Data, Set Attribute, and Forward

The listStudents() method performs the three essential steps for displaying the view: retrieving data, storing it in the request scope, and forwarding to the JSP view:

```
private void listStudents(HttpServletRequest request, HttpServletResponse
response)

throws ServletException, IOException {

    List<Student> students = studentDAO.getAllStudents();

    request.setAttribute("students", students);

    RequestDispatcher dispatcher =
request.getRequestDispatcher("/views/student-list.jsp");

    dispatcher.forward(request, response);

}
```

Task 2.2: Add More CRUD Methods (10 points)

Add these methods to StudentController:

Required Methods:

1. `showNewForm()` - Display add form
2. `showEditForm()` - Display edit form
3. `doPost()` - Handle POST requests
4. `insertStudent()` - Add new student
5. `updateStudent()` - Update student

6. `deleteStudent()` - Delete student

Evaluation Criteria:

Criteria	Points
<code>showNewForm()</code> forwards to form	2
<code>showEditForm()</code> loads student and forwards	2
<code>doPost()</code> routes correctly	2
<code>insertStudent()</code> creates student	2
<code>updateStudent()</code> modifies student	1
<code>deleteStudent()</code> removes student	1

Checkpoint #2: Show instructor that all controller methods work.

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

    String action = request.getParameter("action");

    switch (action) {
        case "insert":
            insertStudent(request, response);
            break;
        case "update":
            updateStudent(request, response);
            break;
    }
}

// List all students
private void listStudents(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {

    List<Student> students = studentDAO.getAllStudents();
    request.setAttribute("students", students);

    RequestDispatcher dispatcher =
request.getRequestDispatcher("/views/student-list.jsp");
    dispatcher.forward(request, response);
}

// Show form for new student
```

```
private void showNewForm(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    RequestDispatcher dispatcher =
request.getRequestDispatcher("/views/student-form.jsp");
    dispatcher.forward(request, response);
}

// Show form for editing student
private void showEditForm(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    int id = Integer.parseInt(request.getParameter("id"));
    Student existingStudent = studentDAO.getStudentById(id);

    request.setAttribute("student", existingStudent);

    RequestDispatcher dispatcher =
request.getRequestDispatcher("/views/student-form.jsp");
    dispatcher.forward(request, response);
}

// Insert new student
private void insertStudent(HttpServletRequest request, HttpServletResponse response)
    throws IOException {

    String studentCode = request.getParameter("studentCode");
    String fullName = request.getParameter("fullName");
    String email = request.getParameter("email");
    String major = request.getParameter("major");

    Student newStudent = new Student(studentCode, fullName, email, major);

    if (studentDAO.addStudent(newStudent)) {
        response.sendRedirect("student?action=list&message=Student added successfully");
    } else {
        response.sendRedirect("student?action=list&error=Failed to add student");
    }
}

// Update student
private void updateStudent(HttpServletRequest request, HttpServletResponse response)
    throws IOException {

    int id = Integer.parseInt(request.getParameter("id"));
    String studentCode = request.getParameter("studentCode");
    String fullName = request.getParameter("fullName");
```

```
String email = request.getParameter("email");
String major = request.getParameter("major");

Student student = new Student(studentCode, fullName, email, major);
student.setId(id);

if (studentDAO.updateStudent(student)) {
    response.sendRedirect("student?action=list&message=Student updated
successfully");
} else {
    response.sendRedirect("student?action=list&error=Failed to update
student");
}

// Delete student
private void deleteStudent(HttpServletRequest request, HttpServletResponse
response)
throws IOException {

    int id = Integer.parseInt(request.getParameter("id"));

    if (studentDAO.deleteStudent(id)) {
        response.sendRedirect("student?action=list&message=Student deleted
successfully");
    } else {
        response.sendRedirect("student?action=list&error=Failed to delete
student");
    }
}
```

EXERCISE 3: VIEW LAYER WITH JSTL (15 points)

Estimated Time: 35 minutes

Task 3.1: Create Student List View (8 points)

File: `WebContent/views/student-list.jsp`

Requirements:

- Add JSTL taglib directive
- No scriptlets (no `<% %>`)
- Use `<c:if>` for messages
- Use `<c:forEach>` for student list
- Use EL `${}` for all data access
- Handle empty list case

Evaluation Criteria:

Criteria	Points
JSTL taglib included	1
No scriptlets used	2
c:if for messages	1
c:forEach for students	2
EL for data access	1
Empty list handled	1

JSTL taglib included

The correct taglib directive for Jakarta EE is included at the top to enable JSTL functionality:

```
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
```

No scriptlets used & EL for data access

No Java code blocks (`<% %>`) are used. All data is accessed using **Expression Language** (`${}`), ensuring clean presentation logic:

```
<td>${student.id}</td>
```

```
<td><strong>${student.studentCode}</strong></td>  
  
<td>${student.fullName}</td>  
  
<td>${student.email}</td>  
  
<td>${student.major}</td>
```

c:if for messages

The JSTL <c:if> tags are used to conditionally display success or error messages based on URL parameters (param):

```
<c:if test="${not empty param.message}">  
    <div class="message success">  
        ✓ ${param.message}  
    </div>  
</c:if>
```

c:forEach for students

The <c:forEach> tag iterates over the list of students set by the Controller (\${students}), rendering the table rows dynamically:

```
<c:forEach var="student" items="${students}">  
    <tr>  
        <td>${student.id}</td>  
        <td><strong>${student.studentCode}</strong></td>  
        <td>${student.fullName}</td>  
        <td>${student.email}</td>  
        <td>${student.major}</td>  
        <td>  
            <div class="actions">  
                <a href="student?action=edit&id=${student.id}" class="btn  
                btn-secondary">  
                     Edit  
                </a>  
                <a href="student?action=delete&id=${student.id}"  
                class="btn btn-danger"  
                onclick="return confirm('Are you sure you want to delete  
                this student?')">  
                     Delete  
                </a>  
            </div>  
        </td>  
    </tr>  
</c:forEach>
```

Empty list handled

A `<c:choose>` structure handles the conditional rendering, displaying the table only if the students list is not empty, and showing an empty state message otherwise:

```
<c:otherwise>
    <div class="empty-state">
        <div class="empty-state-icon">👤</div>
        <h3>No students found</h3>
        <p>Start by adding a new student</p>
    </div>
</c:otherwise>
```

Task 3.2: Create Student Form View (7 points)

File: `WebContent/views/student-form.jsp`

Requirements:

- Single form for both Add and Edit
- Use `<c:choose>` for dynamic title
- Use `<c:if>` to show/hide fields conditionally
- Pre-fill values for edit mode
- No scriptlets

Evaluation Criteria:

Criteria	Points
c:choose for title	1
c:if for heading	1
Dynamic action field	1
Conditional id field	1
Pre-filled values	2
No scriptlets	1

Checkpoint #3: Show instructor complete MVC application working.

c:choose for title and c:if for heading

The file uses <c:choose> to dynamically set the page title and the main heading based on the presence of the student object in the request scope:

```
<c:choose>
  <c:when test="\${student != null}">
     Edit Student
  </c:when>
  <c:otherwise>
     Add New Student
  </c:otherwise>
</c:choose>
```

Dynamic action field

A hidden input field correctly determines the submission action (update or insert) using a simple EL ternary operator, eliminating the need for separate forms:

```
<input type="hidden" name="action"
       value="\${student != null ? 'update' : 'insert'}">
```

Conditional id field

The hidden id field, which is required only for updating an existing record, is correctly included using a <c:if> tag:

```
<c:if test="\${student != null}">
  <input type="hidden" name="id" value="\${student.id}">
</c:if>
```

Pre-filled values and No scriptlets

All input fields are pre-filled using EL (`${student.attribute}`), and the selection for the major dropdown is correctly set using a ternary operator within the option tag. No scriptlets (`<% %>`) were used:

```
<option value="Computer Science"
${student.major == 'Computer Science' ? 'selected' : ''}>
    Computer Science
</option>
```

EXERCISE 4: COMPLETE CRUD OPERATIONS (10 points)

Estimated Time: 20 minutes

Task 4.1: Complete DAO Methods (5 points)

Add remaining methods to `StudentDAO.java`:

Evaluation Criteria:

Criteria	Points
getStudentById works	1
addStudent works	1.5
updateStudent works	1.5
deleteStudent works	1

```
// Get student by ID
public Student getStudentById(int id) {
    String sql = "SELECT * FROM students WHERE id = ?";

    try (Connection conn = getConnection();
        PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setInt(1, id);
        ResultSet rs = pstmt.executeQuery();

        if (rs.next()) {
            Student student = new Student();
            student.setId(rs.getInt("id"));
            student.setStudentCode(rs.getString("student_code"));
            student.setFullName(rs.getString("full_name"));
            student.setEmail(rs.getString("email"));
            student.setMajor(rs.getString("major"));
            student.setCreatedAt(rs.getTimestamp("created_at"));
            return student;
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return null;
}

// Add new student
public boolean addStudent(Student student) {
    String sql = "INSERT INTO students (student_code, full_name, email, major)
VALUES (?, ?, ?, ?)";

    try (Connection conn = getConnection();
        PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setString(1, student.getStudentCode());
        pstmt.setString(2, student.getFullName());
        pstmt.setString(3, student.getEmail());
        pstmt.setString(4, student.getMajor());
        int rowsAffected = pstmt.executeUpdate();

        if (rowsAffected > 0) {
            return true;
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

```

try (Connection conn = getConnection();
     PreparedStatement pstmt = conn.prepareStatement(sql)) {

    pstmt.setString(1, student.getStudentCode());
    pstmt.setString(2, student.getFullName());
    pstmt.setString(3, student.getEmail());
    pstmt.setString(4, student.getMajor());

    int rowsAffected = pstmt.executeUpdate();
    return rowsAffected > 0;

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

// Update student
public boolean updateStudent(Student student) {
    String sql = "UPDATE students SET student_code = ?, full_name = ?, email = ?,
    major = ? WHERE id = ?";

    try (Connection conn = getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setString(1, student.getStudentCode());
        pstmt.setString(2, student.getFullName());
        pstmt.setString(3, student.getEmail());
        pstmt.setString(4, student.getMajor());
        pstmt.setInt(5, student.getId());

        int rowsAffected = pstmt.executeUpdate();
        return rowsAffected > 0;

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

// Delete student
public boolean deleteStudent(int id) {
    String sql = "DELETE FROM students WHERE id = ?";

    try (Connection conn = getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setInt(1, id);
        int rowsAffected = pstmt.executeUpdate();
        return rowsAffected > 0;

    } catch (SQLException e) {

```

```
        e.printStackTrace();
        return false;
    }
}
```

Scenario: User clicks the "Edit" link for a specific student.

1. Browser → GET /student?action=edit&id=5
2. Controller receives request
3. Extracts action=edit and id=5 parameters
4. Calls StudentDAO.getStudentById(5)
5. DAO executes SELECT * FROM students WHERE id = ?
6. Maps ResultSet to a single Student object
7. Returns the Student object
8. Controller sets the Student object as a request attribute
9. Forwards to student-form.jsp
10. View (JSP) populates the form fields using the student's data

```
<input type="text" name="fullName" value="${student.fullName}">
```

11. Browser receives HTML response with the pre-filled form

The screenshot shows a modal dialog box titled "Edit Student". The dialog has a purple header bar with the title and a white content area. Inside, there are four input fields with validation messages and a dropdown menu. At the bottom are two buttons: "Update Student" and "Cancel".

Field	Value	Description
Student Code *	SV004	Format: 2 letters + 3+ digits
Full Name *	Pham Anh D	
Email *	D@gmail.com	
Major *	Computer Science	

Edit Student

Student Code *

SV004

Format: 2 letters + 3+ digits

Full Name *

Pham Anh D

Email *

D@gmail.com

Major *

Computer Science

Update Student Cancel

Scenario: User fills out the "Add New Student" form and clicks "Save".

1. Browser → POST /student?action=insert (with form data in the body)
2. Controller receives request
3. Creates a new Student object from form parameters
4. Calls StudentDAO.addStudent(newStudent)
5. DAO executes INSERT INTO students (...) VALUES (...)
6. Returns true on success
7. Controller checks the boolean result and redirects
8. Browser receives a redirect response to GET /student?action=list

Student Management System

MVC Pattern with Jakarta EE & JSTL

✓ Student added successfully

[+ Add New Student](#)

ID	STUDENT CODE	FULL NAME	EMAIL	MAJOR	ACTIONS
4	SV004	Pham Anh D	D@gmail.com	Computer Science	Edit Delete
3	SV003	Le Van C	c.le@example.com	Software Engineering	Edit Delete
2	SV002	Tran Thi B	b.tran@example.com	Information Technology	Edit Delete
1	SV001	Nguyen Van A	a.nguyen@example.com	Computer Science	Edit Delete

Scenario: User modifies data on the "Edit Student" form and clicks "Update".

1. Browser → POST /student?action=update (with form data, including student ID)
2. Controller receives request
3. Creates a Student object (with ID) from form parameters
4. Calls StudentDAO.updateStudent(updatedStudent)
5. DAO executes UPDATE students SET ... WHERE id = ?
6. Returns true on success
7. Controller checks the boolean result and redirects
8. Browser receives a redirect response to GET /student?action=list

Student Management System

MVC Pattern with Jakarta EE & JSTL

Student updated successfully

[Add New Student](#)

ID	STUDENT CODE	FULL NAME	EMAIL	MAJOR	ACTIONS
4	SV004	Pham Hoang P	D@gmail.com	Computer Science	 Edit Delete
3	SV003	Le Van C	c.le@example.com	Software Engineering	 Edit Delete
2	SV002	Tran Thi B	b.tran@example.com	Information Technology	 Edit Delete
1	SV001	Nguyen Van A	a.nguyen@example.com	Computer Science	 Edit Delete

Scenario: User clicks the "Delete" button for a specific student.

1. Browser → GET /student?action=delete&id=5
2. Controller receives request
3. Extracts action=delete and id=5 parameters
4. Calls StudentDAO.deleteStudent(5)
5. DAO executes DELETE FROM students WHERE id = ?
6. Returns true on success
7. Controller checks the boolean result and redirects
8. Browser receives a redirect response to GET /student?action=list

Student Management System

MVC Pattern with Jakarta EE & JSTL

✓ Student deleted successfully

[+ Add New Student](#)

ID	STUDENT CODE	FULL NAME	EMAIL	MAJOR	ACTIONS
3	SV003	Le Van C	c.le@example.com	Software Engineering	>Edit Delete
2	SV002	Tran Thi B	b.tran@example.com	Information Technology	Edit Delete
1	SV001	Nguyen Van A	a.nguyen@example.com	Computer Science	Edit Delete