International University

School of Computer Science and Engineering

Web Application Development

Laboratory

IT093IU

Lab #4 Practice

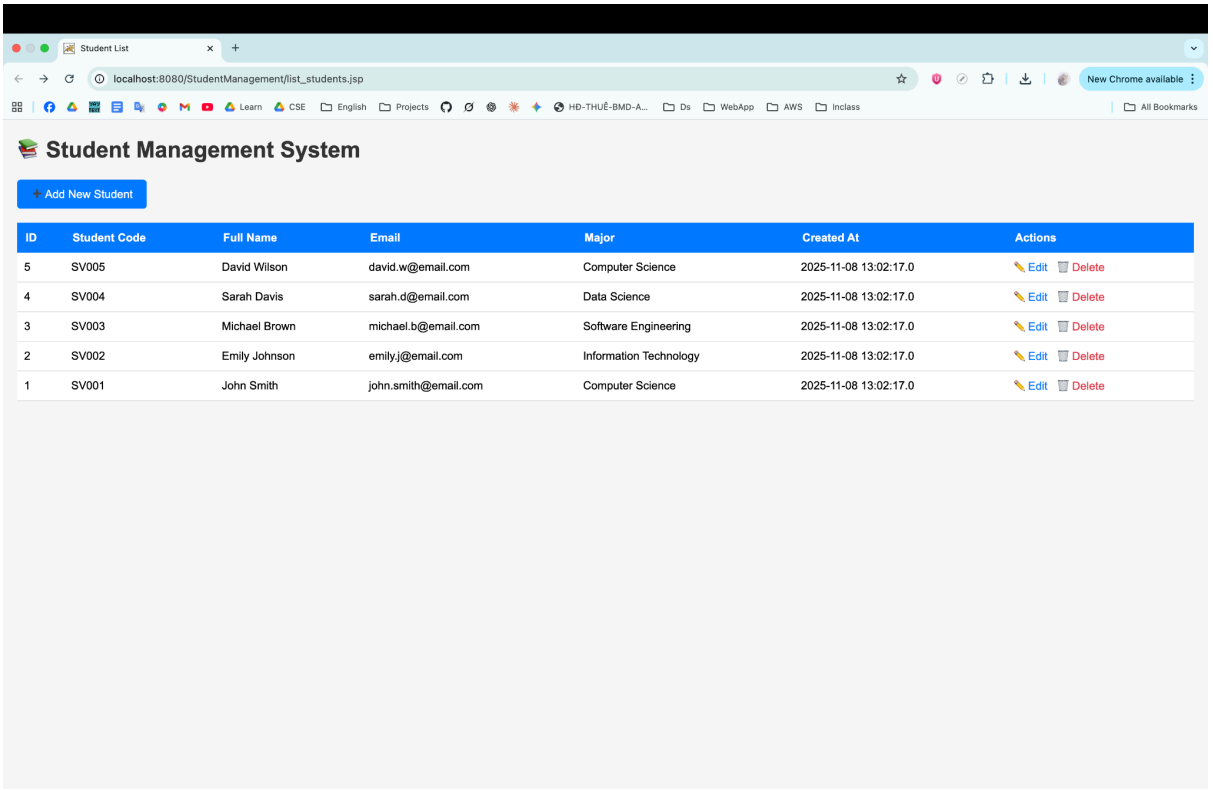
Submitted by

Nguyễn Hồng Ngọc Hân - ITCSIU22229

EXERCISE 1: SETUP AND DISPLAY

Task 1.1: Project Setup

Task 1.2: Display Student List



Explain:

When a client sends a GET request to list_students.jsp, the web server (Tomcat) locates this JSP file in the project directory. The JSP is then compiled into a servlet and executed. Inside the code, it first checks for any message or error parameters in the request to display status notifications. Next, it connects to the MySQL database using JDBC (DriverManager.getConnection()), and executes the SQL query SELECT * FROM students ORDER BY id DESC to retrieve all student records. The server processes the results with a loop (while (rs.next())) and dynamically generates HTML table rows for each student's data. After rendering, the servlet sends the completed HTML output back as a response to the client's browser, which displays the formatted table with the student list, including edit and delete links.

EXERCISE 2: CREATE OPERATION

Task 2.1: Create Add Student Form

Explain:

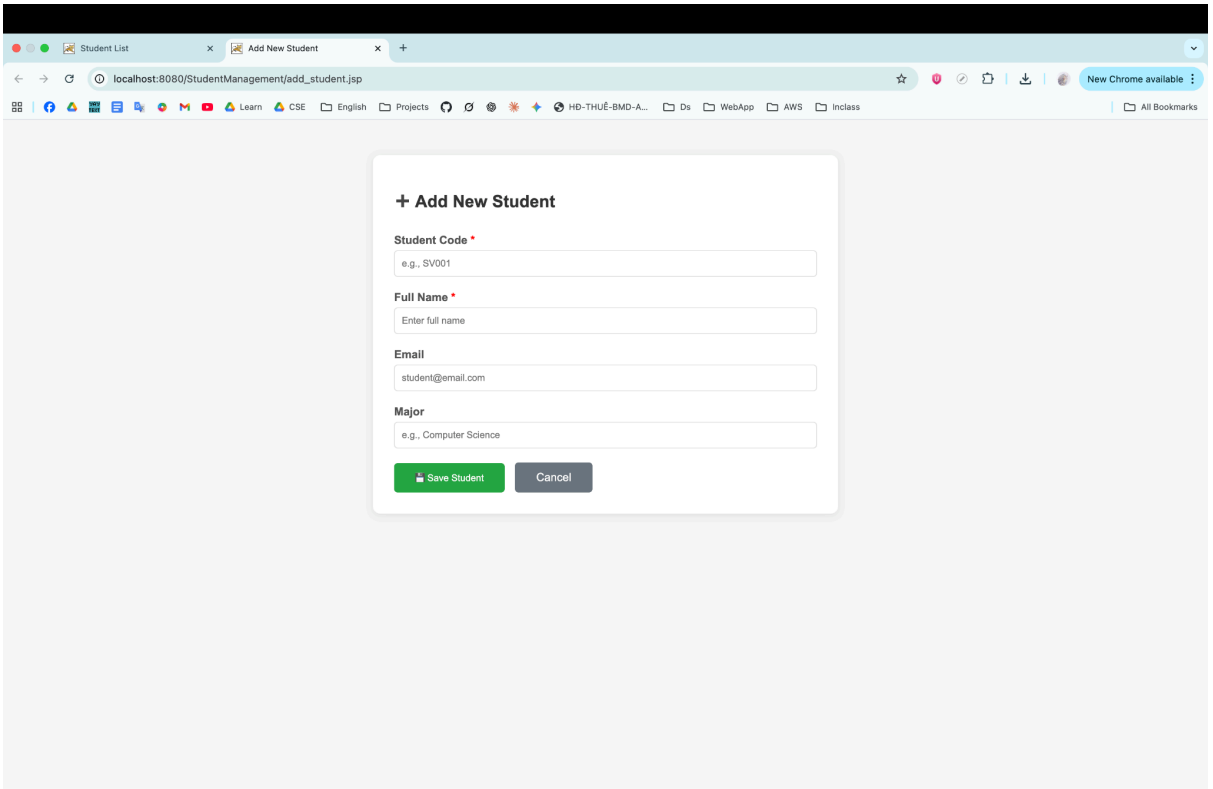
This page provides a clean HTML form that allows the user to input new student details. It includes fields for Student Code, Full Name, Email, and Major, with the first two marked as required using the required attribute. The pattern attribute ensures that the student code follows a specific format (two uppercase letters and at least three digits). When the user clicks Save Student, the form data is sent to

process_add.jsp through a POST request. If a required field is missing, the browser prevents submission and displays an inline validation message. The design of a Cancel button redirects back to list_students.jsp.

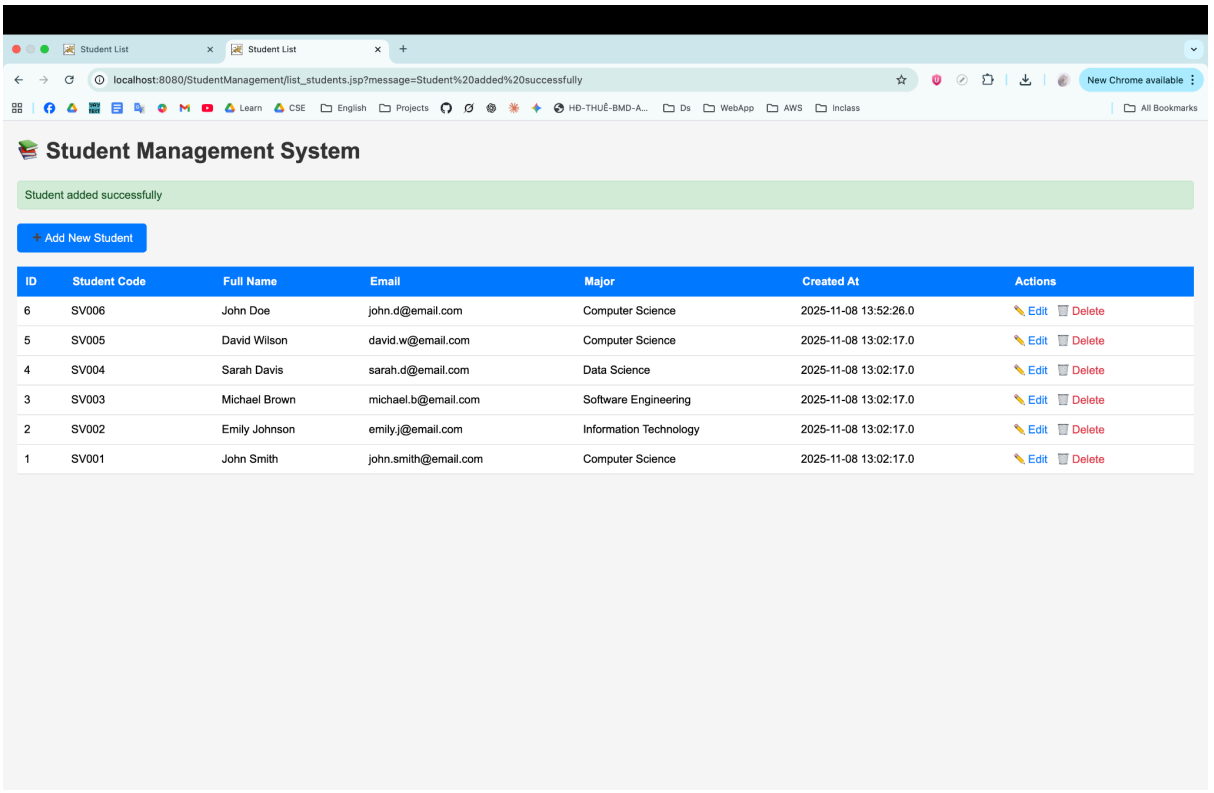
Task 2.2: Process Add Student

Explain:

This file handles the backend logic for inserting a new student record into the database. When the user submits the form, the JSP retrieves form data using request.getParameter(). It checks whether required fields are empty; if yes, it redirects back to add_student.jsp with the error message “Required fields are missing.” Next, it loads the MySQL JDBC driver, connects to the student_management database, and prepares an SQL INSERT statement using PreparedStatement to prevent SQL injection. If the insertion is successful, it redirects to list_students.jsp with a success message. If an error occurs (for example, a duplicate student code), it catches the exception and redirects with an appropriate error message. Finally, the code closes all database resources to avoid connection leaks.



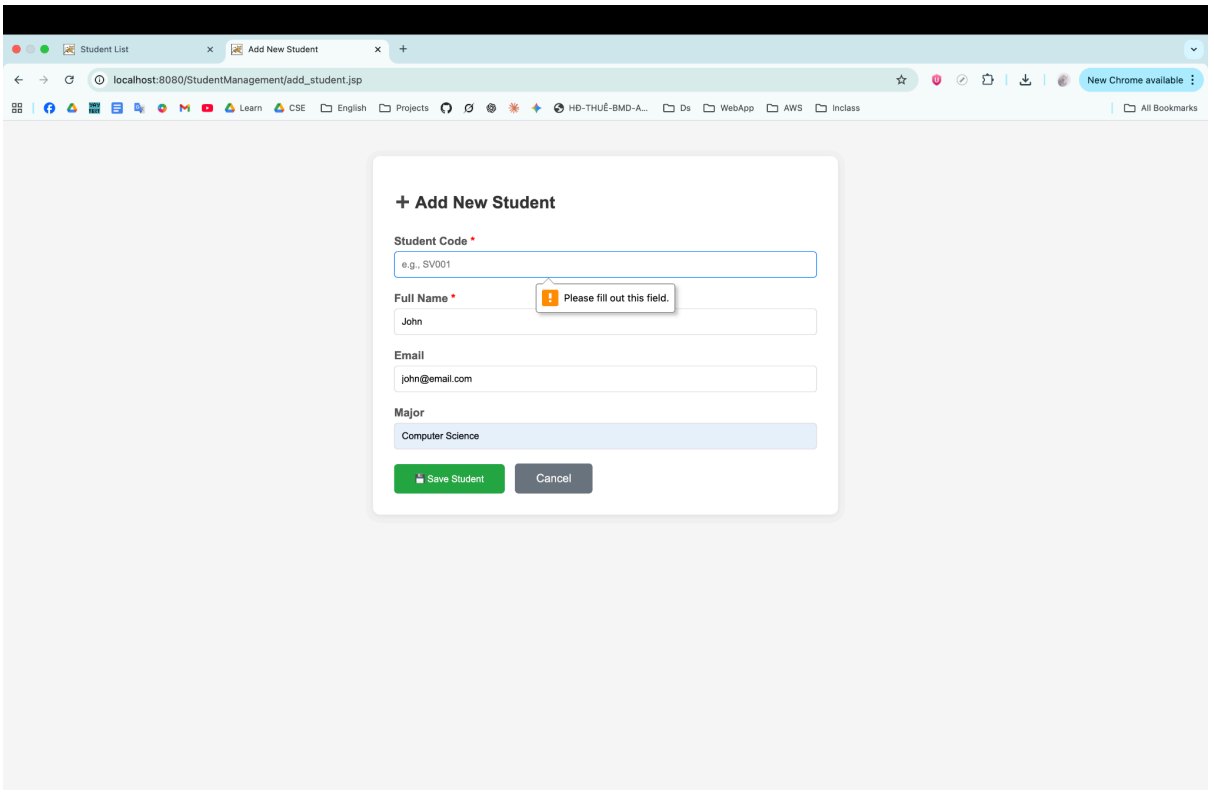
- **Test Cases 1:**



Explain:

In this case, the user opens the Add Student form and fills all the fields with valid information, for example, Student Code: SV006, Full Name: John Doe, Email: john@email.com, and Major: Computer Science. After clicking Save Student, the browser sends a POST request to process_add.jsp, where the system validates the inputs, connects to the database, and runs an INSERT command. Once the data is successfully added, the code redirects the user to list_students.jsp using response.sendRedirect("list_students.jsp?message=Studentadded successfully";. The user sees a confirmation message, “Student added successfully,” and the new record appears in the student table.

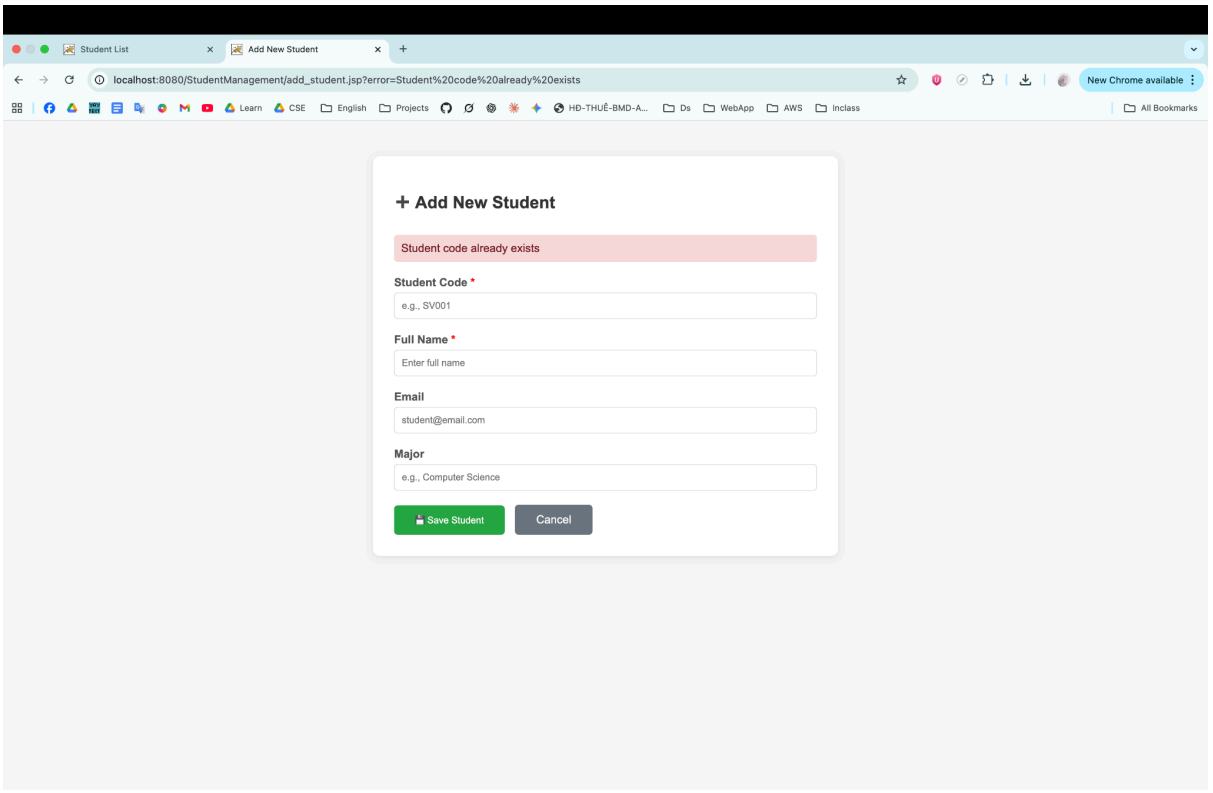
- **Test Cases 2:**



Explain:

In this scenario, the user opens the Add Student form but leaves one of the required fields empty; for instance, Student Code is blank while the Full Name is filled as John. When the user clicks Save Student, the form data is sent to process_add.jsp, where the system performs a null or empty check for required fields. Since student_code is missing, the condition triggers a redirect using response.sendRedirect("add_student.jsp?error=Required fields are missing");.The browser then reloads the Add Student form and displays a red warning message: “Required fields are missing.” The database remains unchanged, ensuring data integrity.

- **Test Cases 3:**



Explain:

In this final test case, the user fills out the form with an existing student code, such as SV001 that already exists in the database. The system executes the INSERT query, but MySQL detects a duplicate key error. The exception message contains the text “Duplicate entry,” which the program checks in the catch block. When detected, it redirects to response.sendRedirect("add_student.jsp?error=Student code already exists");. The user is redirected back to the Add Student form with an error message displayed in red. The duplicate record is not inserted, and the existing data remains intact.

EXERCISE 3: UPDATE OPERATION

Task 3.1: Create Edit Form

Explain:

This page allows users to update an existing student’s information. When a user clicks the Edit button on the student list page, the browser sends a GET request to edit_student.jsp?id=.... The JSP first reads the id parameter and validates it. If it is missing or invalid, the system redirects to list_students.jsp with an error message. Once validated, the program connects to the MySQL database using JDBC and executes the query "SELECT * FROM students WHERE id = ?" to retrieve the student’s current data. The retrieved values are then pre-filled into an editable HTML form. The Student Code field is marked as read-only to prevent changes to the primary identifier. The form’s design matches the “Add Student” layout for

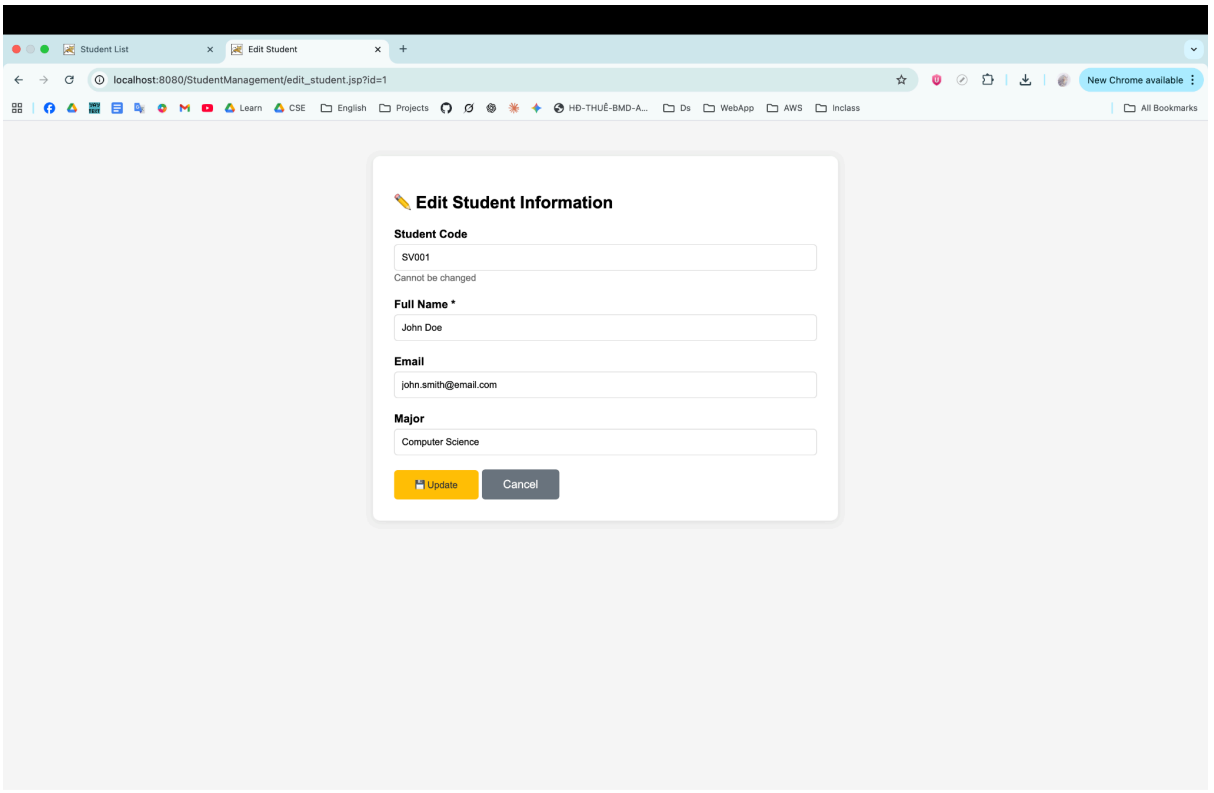
consistency, and clicking Update sends the modified data to process_edit.jsp via the POST method for processing.

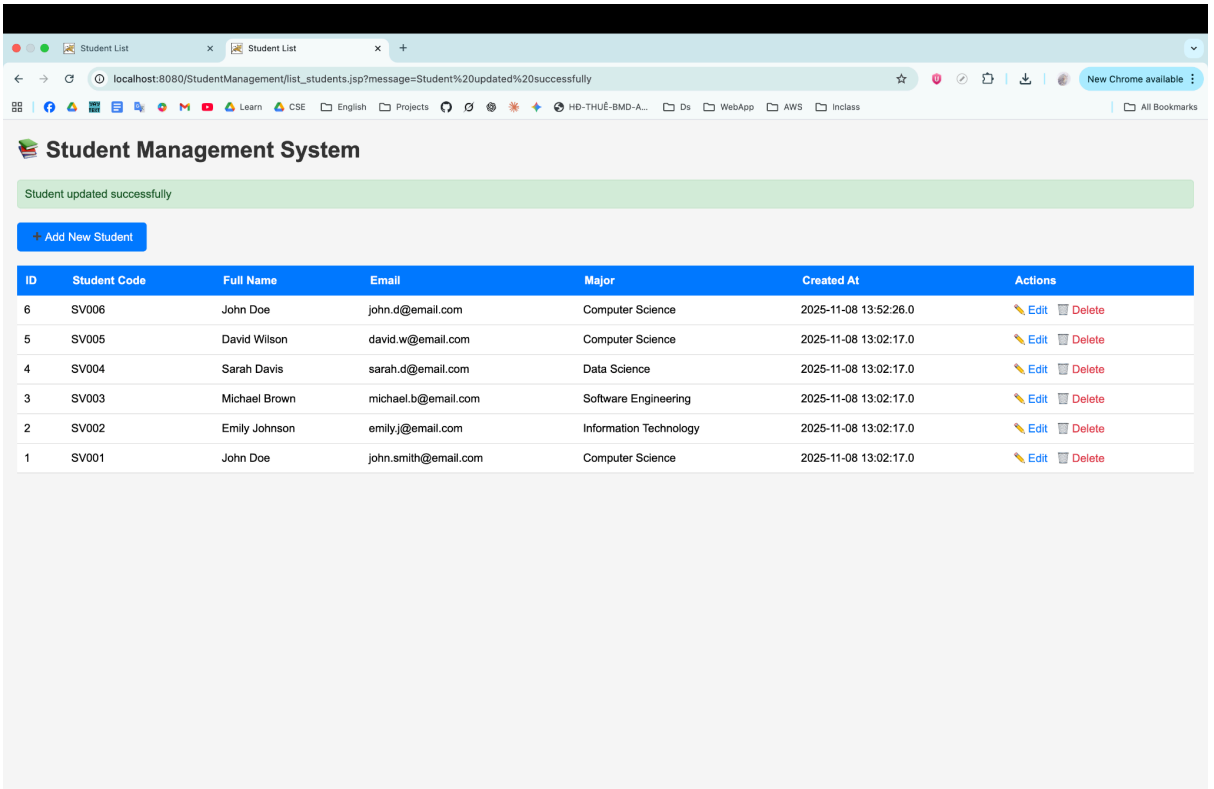
Task 3.2: Process Update

Explain:

This file receives the submitted form data and performs the actual update operation in the database. It retrieves values using request.getParameter() and validates that the ID and full name are not null or empty. If validation fails, it redirects with an error message. After validation, the code loads the MySQL JDBC Driver, connects to the database, and prepares the SQL statement "UPDATE students SET full_name=?, email=?, major=? WHERE id=?" using a PreparedStatement to prevent SQL injection. The parameters are bound to the updated values, and executeUpdate() is called. If at least one record is modified, the page redirects to the student list with the message “Student updated successfully.” If not, or if an exception occurs, the user is redirected back to the edit page with an appropriate error message. Finally, all database resources are closed in the finally block.

- **Test case 1:**



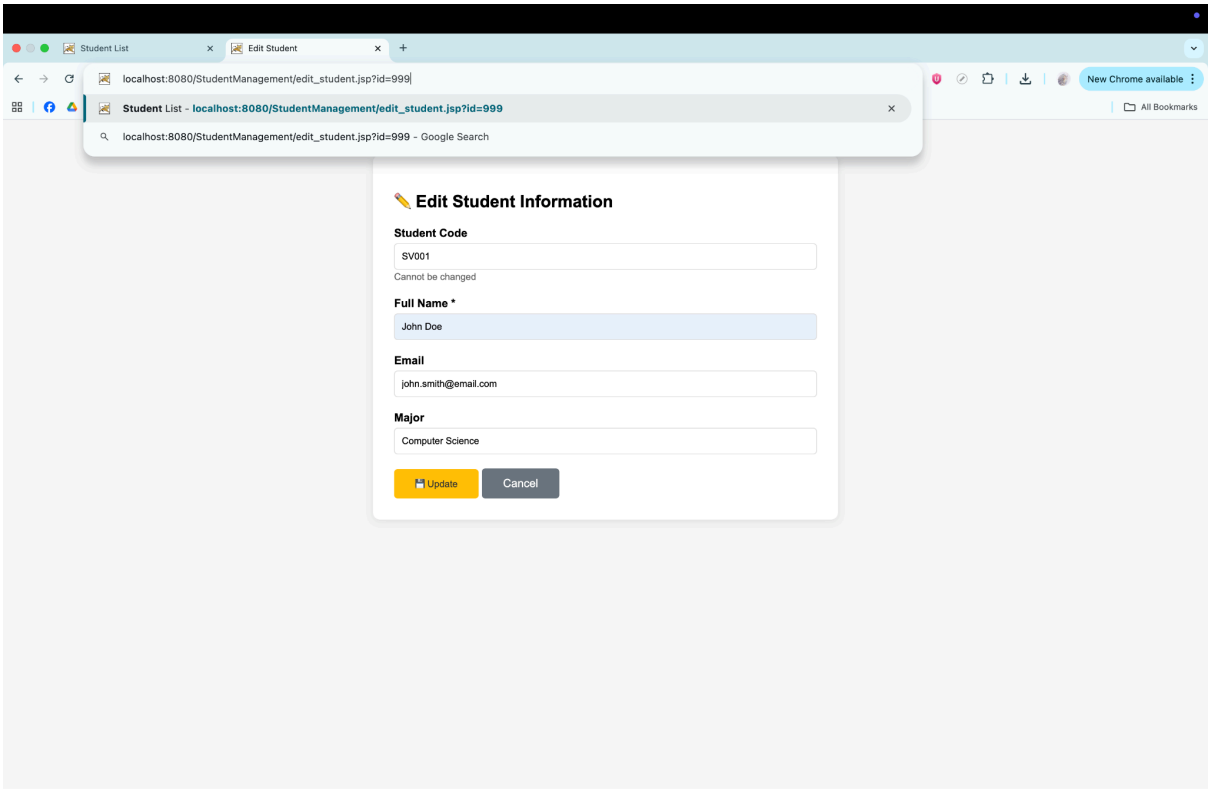


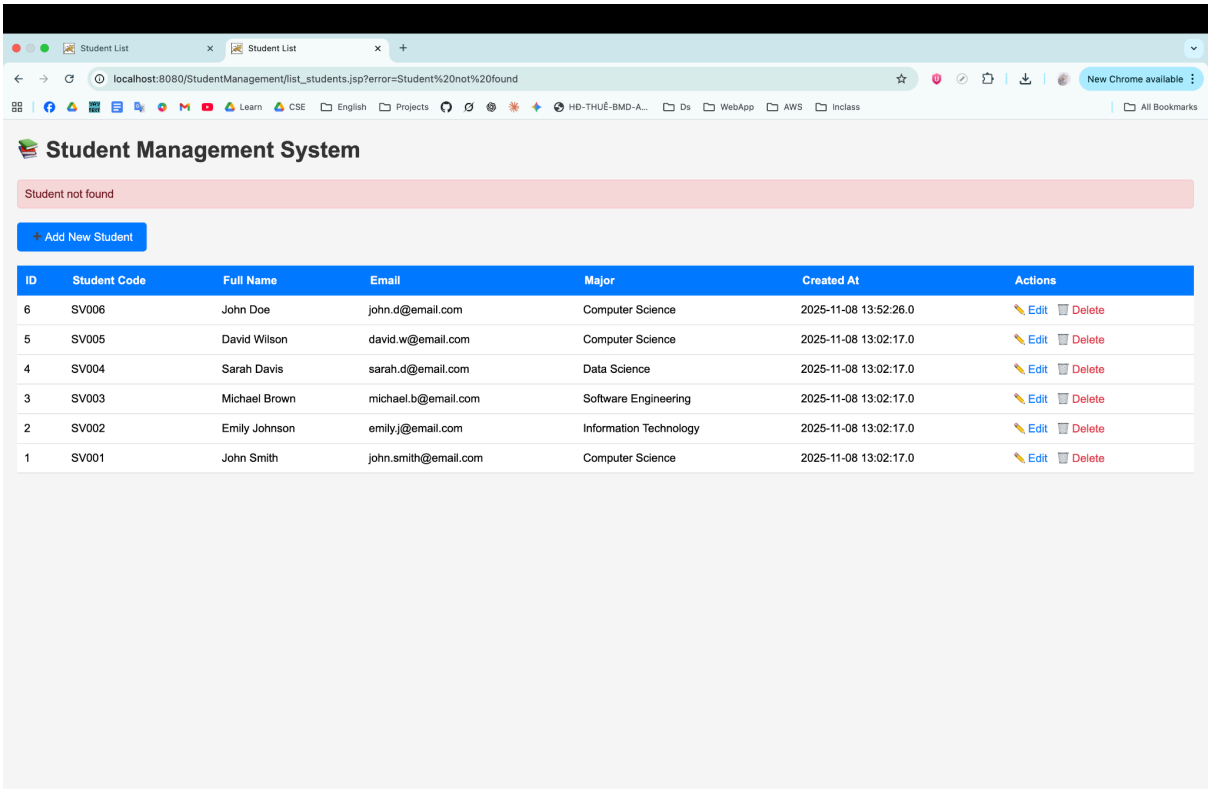
Explain:

In this case, the user opens the edit form for a student named “John Smith” and changes the name to “John Doe.” After pressing Update, the form sends the updated information to process_edit.jsp through a POST request. The server validates the

inputs and executes the SQL statement UPDATE students SET full_name=?, email=?, major=? WHERE id=?. Since the data is valid and the record exists, the update executes successfully, and the system redirects to the student list page. A green success message appears saying “Student updated successfully,” and the table now shows the updated name “John Doe.” This confirms that the update function works correctly for valid data.

- **Test Cases 2:**



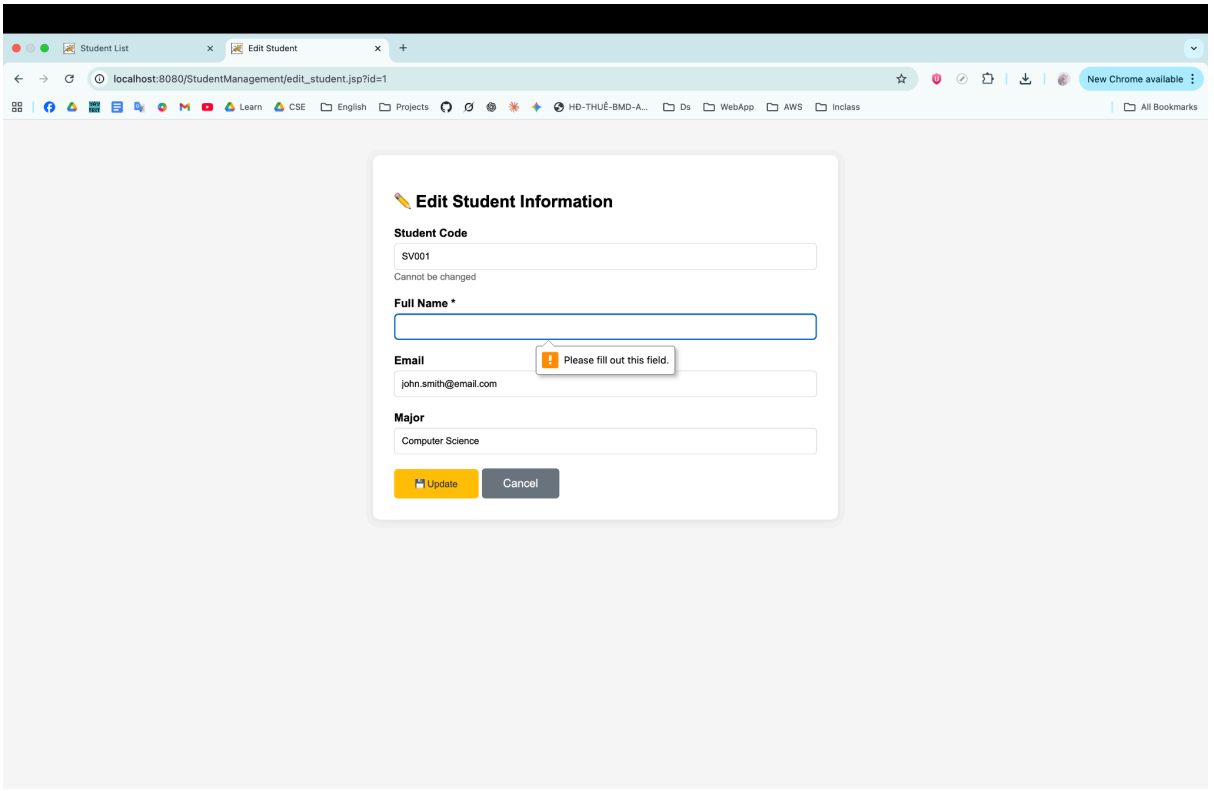


Explain:

In this test, the user tries to access edit_student.jsp?id=999, where ID 999 does not exist in the database. When the JSP executes the query SELECT * FROM students WHERE id=?, no record is found, and the system immediately redirects back to the student list page. A red error message, “Student not found” appears at the top of the

list page. This prevents users from attempting to edit non-existent records, ensuring data consistency and preventing unnecessary database operations.

- **Test Cases 3:**



Explain:

In this situation, the user opens an existing student's edit page but leaves the Full Name field blank before clicking Update. When the form is submitted, process_edit.jsp retrieves the parameters and checks for empty required fields. Since the full_name field is missing, the system triggers validation failure and redirects the user to the list page with an error message, "Required field missing." The SQL update is not executed, and the original data in the database remains unchanged. This confirms that the application correctly enforces input validation before performing database updates, ensuring data quality and system reliability.

EXERCISE 4: DELETE OPERATION

Task 4.1: Implement Delete.

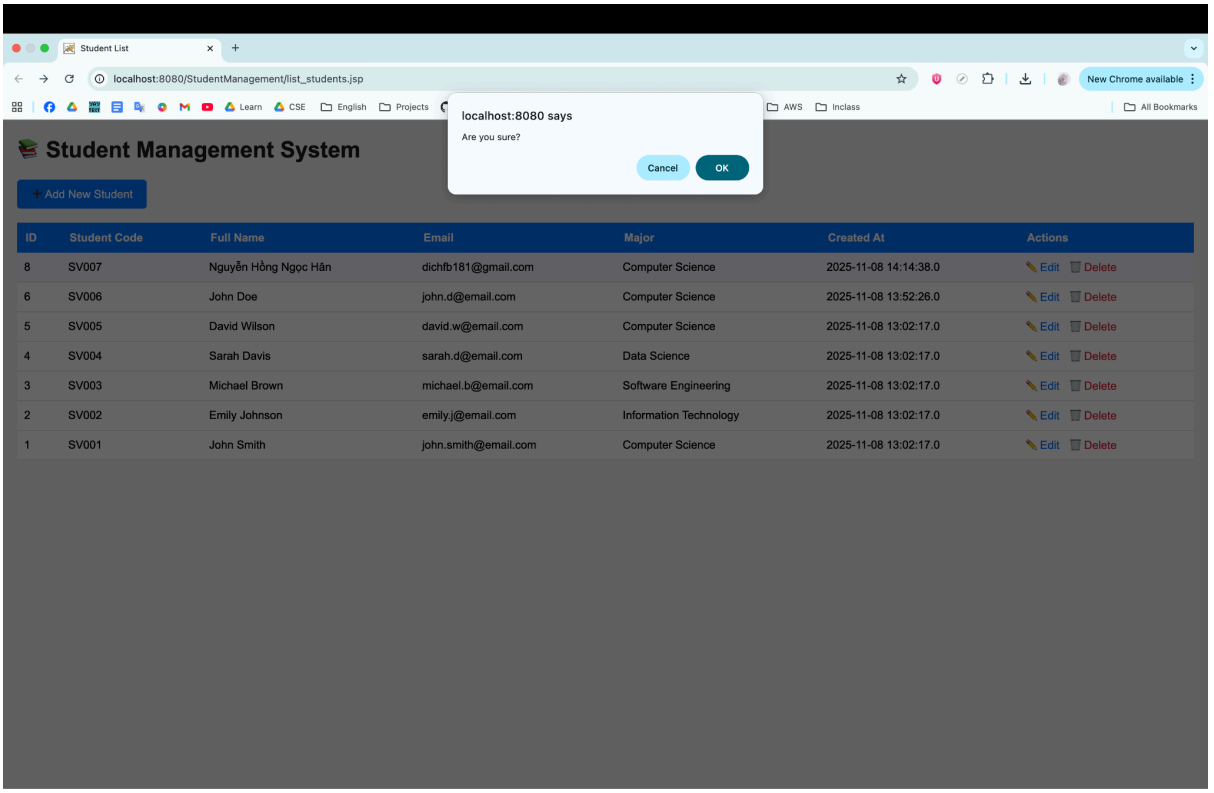
Task 4.2: Add Delete Links and Confirmation.

Explain:

In this task, the system enables secure deletion of student records through both backend logic and user confirmation. Each record in the student list includes a delete link like "<a href="delete_student.jsp?id=<%=id%>" onclick="return confirm('Are you sure?')">" which triggers a confirmation dialog before deletion. When the user clicks Delete, the browser asks for confirmation; choosing Cancel stops the process, while

choosing OK sends a GET request to delete_student.jsp?id=.... On the server side, the code retrieves the student ID, validates it, and connects to the student_management database via JDBC. It then executes a prepared SQL statement, "DELETE FROM students WHERE id = ?" to remove the record safely. If deletion is successful, the user is redirected to the list page with the message “Student deleted successfully.” If the ID is invalid, missing, or linked to another table, the program handles these cases gracefully with appropriate error messages. This combination of client-side confirmation and server-side validation ensures both data integrity and user safety in the deletion process.

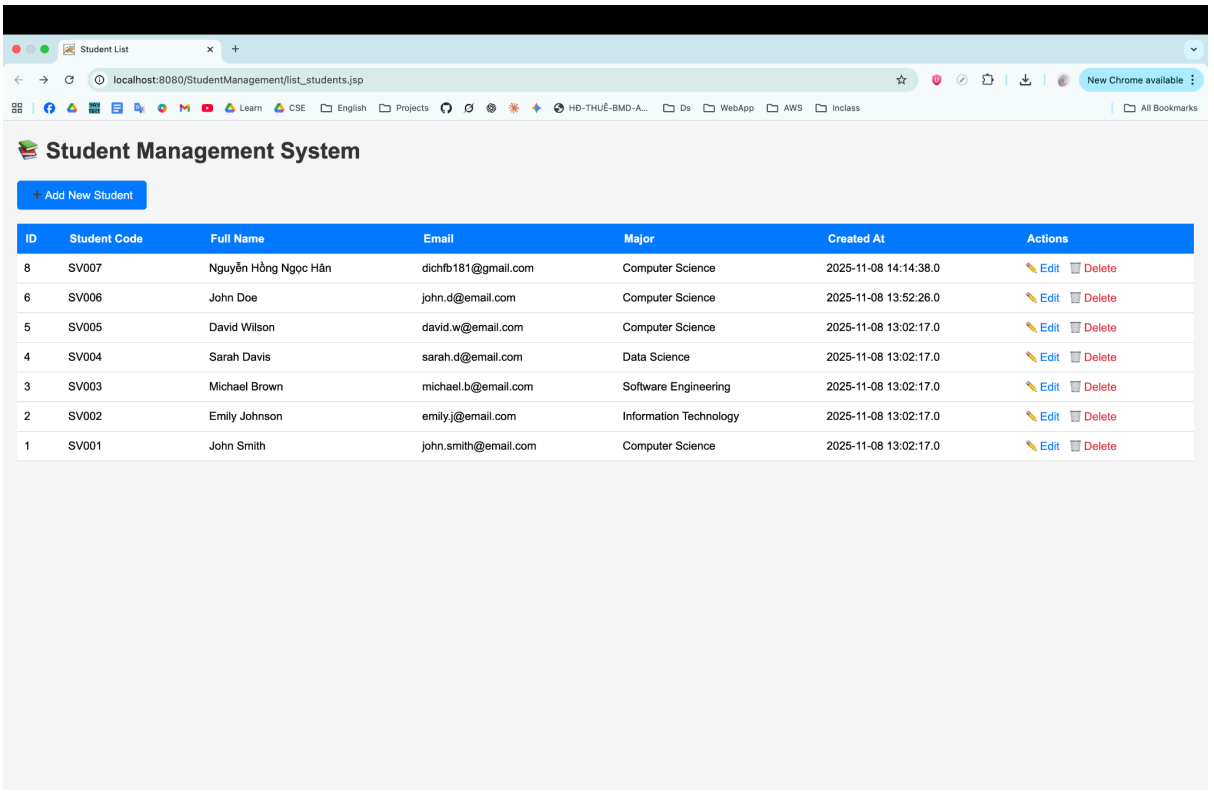
- **Test case 1: Click delete on test student**



Explain:

In this test, the user selects a test student record from the list and clicks the Delete link. The system displays a confirmation dialog. After the user confirms, the browser sends a GET request to delete_student.jsp?id=<studentId>. The server reads the ID, connects to the MySQL database, and executes "DELETE FROM students WHERE id = ?" Since the record exists, one row is deleted (rowsAffected > 0). The code then redirects the browser back to list_students.jsp with the message “Student deleted successfully.” When the list reloads, the deleted record no longer appears, proving the deletion worked correctly.

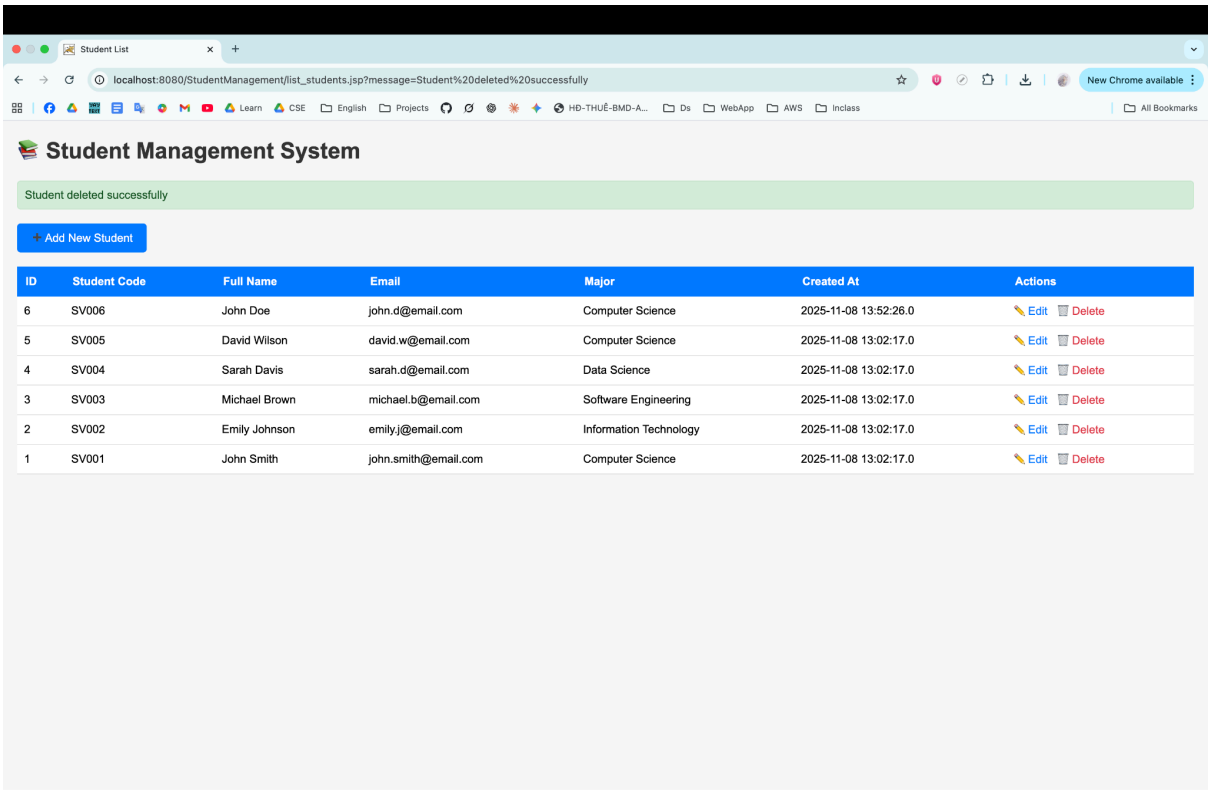
- **Test case 2: Cancel in dialog → Student should remain**



Explain:

In this scenario, the user clicks the Delete icon but chooses Cancel when the browser asks for confirmation. Because the function return confirm('Are you sure?') evaluates to false, the link action is stopped before sending any request to the server. Therefore, no communication occurs with delete_student.jsp, and the database remains untouched. When the list is refreshed, the same student record still appears, confirming that the cancel option effectively prevents accidental deletions.

- **Test case 3: Confirm in dialog → Student should be deleted**



Explain:

Here, the user clicks the Delete button and chooses OK in the confirmation popup. The browser then sends the request to delete_student.jsp?id=<studentId>. The JSP retrieves the ID, establishes a database connection, and executes the prepared DELETE command. Since the student ID exists, the query affects one row and triggers response.sendRedirect("list_students.jsp?message=Student deleted successfully");. The user is redirected to the student list page with a green success notification. The deleted student's record no longer appears in the table, indicating that the deletion and page redirection worked as expected.