



<b>Course Code: SL3001</b>	<b>Course: Software Development and construction</b>
<b>Instructor:</b>	<b>Yasir Arfat</b>

### Lab # 12

#### Objective

In this lab, students will first learn about the Model-View-Controller (MVC) framework, a design pattern that separates the application logic into distinct components for better organization and maintainability. Following this, they will build a Spring Boot application to perform CRUD (Create, Read, Update, Delete) operations, integrating it with Oracle SQL Developer as the database using the ojdbc11.jar driver. Additionally, the lab will include implementing user authentication with session and cookies to manage user login states securely. This lab provides hands-on experience in setting up a Spring Boot project, understanding the MVC structure, and implementing fundamental web application functionality.

#### What is the MVC Framework?

The **Model-View-Controller (MVC)** framework is a design pattern used to develop applications by dividing the program logic into three interconnected components: **Model**, **View**, and **Controller**. This separation of concerns helps in organizing the code, enhancing maintainability, scalability, and testability.

#### Components of MVC:

1. **Model:**
  - Represents the application's data, business logic, and rules.

- Responsible for managing and updating data.
- Example: A class representing an employee (Employee) with fields like id, name, and department.

## 2. **View:**

- Represents the user interface (UI).
- Displays the data to the user and captures user inputs.
- Example: An HTML page showing a list of employees or a form for adding a new employee.

## 3. **Controller:**

- Acts as an intermediary between the Model and the View.
- Handles user inputs, processes them (using the Model), and updates the View accordingly.
- Example: A Spring Boot controller handling requests like "add a new employee" or "delete an employee."

### How MVC Works:

- **User Interaction:** The user interacts with the **View** (e.g., submits a form or clicks a button).
- **Controller Processing:** The **Controller** processes the user request, interacts with the **Model** to fetch or modify data, and updates the **View**.
- **Data Handling:** The **Model** contains the logic to handle data, such as retrieving it from a database or performing calculations.
- **View Update:** The **View** is updated based on the results from the **Controller** and presented to the user.

### Spring Boot Crud

#### 1. Setting Up database

In this CRUD lab, we will work with an employee table in Oracle SQL Developer, which will have the following structure:

```
CREATE TABLE employee (  
  
    id NUMBER,  
  
    name VARCHAR2(100),  
  
    email VARCHAR2(100),  
  
    password VARCHAR2(100),  
  
    PRIMARY KEY (id)  
  
);
```

Since Oracle does not support auto-increment for primary keys, we will use a sequence and a trigger to automatically generate unique id values when inserting new records. Here's how we'll handle this:

#### **Sequence:**

A sequence generates a unique number to be used as the employee id. The sequence starts with 1 and increments by 1.

```
CREATE SEQUENCE emp_seq  
  
START WITH 1  
  
INCREMENT BY 1  
  
NOCACHE
```

#### **Trigger:**

A trigger is created to automatically assign the next value from the sequence (emp\_seq.NEXTVAL) to the id field when a new record is inserted into the employee table. This ensures that the id is automatically populated if the user doesn't provide a value for it.

```
CREATE OR REPLACE TRIGGER emp_trigger  
  
BEFORE INSERT ON employee  
  
FOR EACH ROW  
  
BEGIN  
  
    IF :NEW.id IS NULL THEN  
  
        SELECT emp_seq.NEXTVAL INTO :NEW.id FROM dual;  
  
    END IF;  
  
END;
```

## 2. Setting Up Project

### Step 1: Create a Gradle Project

- Open IntelliJ IDEA.
- Create a new Gradle project:
- Select File > New > Project.
- Choose Gradle as the build tool.
- Select Java as the language.
- Name the project: employee-crud.
- Set the Group ID and Artifact ID (for example, com.example and employee-crud).

### Step 2: Add Dependencies in build.gradle and Include ojdbc11.jar

1. **Create libs Directory:** Inside your project (e.g., employee-crud), create a folder named libs where you will add the ojdbc11.jar file. The directory structure should look like this:

### Step 3: Configure `application.properties`

In your `src/main/resources/application.properties` file, add the following configuration:

```
spring.application.name=employee-crud
server.port=8085

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=java_crud
spring.datasource.password=fast
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver

# Hibernate configuration
spring.jpa.database-platform=org.hibernate.dialect.OracleDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

### Step 4: Add Dependencies in `build.gradle`

In your `build.gradle` file, add the following dependencies under the dependencies section:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    // For JPA and database operations
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    // For Thymeleaf templating engine
    implementation 'org.springframework.boot:spring-boot-starter-web' // For
building web applications
    developmentOnly 'org.springframework.boot:spring-boot-devtools' // For
hot reloading during development
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    // For testing purposes
    implementation files('libs/ojdbc11.jar') // Oracle JDBC driver (from
libs folder)
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher' // For
running JUnit tests
}
```

### Step 5: Create the model Package and Add the Employee Entity

Now, let's create a package named `model` under `com.example.employeecrud` and add the `Employee` entity class. This will represent the data structure for the employees and be used for CRUD operations.

### Create the `model` Package:

In your project structure, create a new package `model` inside `com.example.employeecrud`:

### Add the `Employee` Class:

Inside the `model` package, create the `Employee.java` file with the following code:

```
package com.example.employeecrud.model;

import jakarta.persistence.*;

@Entity
@Table(name = "employee")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto-increment ID
    private int id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String email;

    @Column(nullable = false)
    private String password;

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

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
```

```

        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}

```

## Explanation of Above code

### EntityClass:

The Employee class is annotated with `@Entity`, making it a JPA entity that will be mapped to the employee table in the Oracle database.

### PrimaryKey(id):

The `@Id` annotation marks the id field as the primary key. The `@GeneratedValue(strategy = GenerationType.IDENTITY)` annotation automatically handles the generation of IDs, although for Oracle, we will rely on the sequence and trigger to generate the ID (instead of auto-generation) as set up earlier.

### ColumnAnnotations:

The `@Column` annotations ensure that the name, email, and password fields are non-null in the database, as required by the table schema.

### GettersandSetters:

The getters and setters allow the Spring Data JPA repository to access and modify the Employee object's properties. These are **necessary for Spring Boot to handle data binding between the entity and the database.**

## Step 5: Create the EmployeeController for CRUD Operations

Now, let's create the `EmployeeController` class that will handle the CRUD operations for the `Employee` entity. This controller is responsible for interacting with the database via the `EmployeeRepository`, and managing the flow between the user interface (HTML pages) and the backend logic.

#### 1. Create the `EmployeeController` class:

In the `com.example.employeecrud.controller` package, create a new class called `EmployeeController.java`.

```
package com.example.employeecrud.controller;

import com.example.employeecrud.model.Employee;
import com.example.employeecrud.repository.EmployeeRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

@Controller
public class EmployeeController {

    @Autowired
    private EmployeeRepository employeeRepository;

    @GetMapping("/addEmployee")
    public String showForm(Model model) {
        model.addAttribute("employee", new Employee());
        return "addEmployee";
    }

    @PostMapping("/addEmployee")
    public String addEmployee(@ModelAttribute Employee employee) {
        employeeRepository.save(employee);
        return "redirect:/employees";
    }

    @GetMapping("/employees")
    public String listEmployees(Model model) {
        model.addAttribute("employees", employeeRepository.findAll());
        return "employeeList";
    }

    @GetMapping("/editEmployee/{id}")
    public String showEditForm(@PathVariable("id") int id, Model model) {
        Employee employee = employeeRepository.findById(id).orElseThrow(() ->
```



```

new IllegalArgumentException("Invalid employee Id:" + id));
    model.addAttribute("employee", employee);
    return "editEmployee";
}

@PostMapping("/editEmployee/{id}")
public String updateEmployee(@PathVariable("id") int id, @ModelAttribute
Employee employee) {
    employee.setId(id);
    employeeRepository.save(employee);
    return "redirect:/employees";
}

@GetMapping("/deleteEmployee/{id}")
public String deleteEmployee(@PathVariable("id") int id) {
    employeeRepository.deleteById(id);
    return "redirect:/employees";
}
}

```

### Explanation of Each Part:

1. **@Controller:**

This annotation marks the class as a Spring MVC controller. It handles HTTP requests and returns a view to the client (e.g., an HTML page).

2. **@Autowired:**

This annotation automatically injects the `EmployeeRepository` into the controller. The repository handles interactions with the database (CRUD operations).

3. **@GetMapping("/addEmployee"):**

This method handles HTTP GET requests for showing the form to add a new employee. It adds an empty `Employee` object to the model, which is then used in the HTML form for binding data.

4. **@PostMapping("/addEmployee"):**

This method handles the form submission for adding a new employee. The `@ModelAttribute` annotation binds the submitted form data to the `Employee` object. After saving the employee to the database, the user is redirected to the employee list page (`/employees`).

5. **@GetMapping("/employees"):**

This method handles HTTP GET requests to list all employees. It fetches all employee records from the database using `employeeRepository.findAll()` and adds them to the model. The `employeeList` view is rendered to display the employees.

6. **@GetMapping("/editEmployee/{id}"):**

This method shows the form to edit an existing employee. It retrieves the employee with the specified id from the database and passes it to the model, where it will be displayed in the edit form.

7. **@PostMapping("/editEmployee/{id}"):**

This method handles the submission of the employee edit form. It updates the employee in the database by setting the employee ID and saving the updated record.

8. **@GetMapping("/deleteEmployee/{id}"):**

This method handles the deletion of an employee. It deletes the employee with the specified id from the database and then redirects the user to the employee list page.

### Why Add This Controller?

- **SeparationofConcerns:**

The controller ensures a clean separation between the presentation layer (HTML pages) and the business logic (interacting with the database via repositories).

- **MappingRequests:**

Each method in the controller maps a specific URL to a certain action (like adding, editing, listing, or deleting an employee). It makes it easy for Spring to handle incoming requests and return appropriate views.

- **CRUDOperations:**

The controller implements all basic CRUD functionality (Create, Read, Update, Delete) for managing employee records in the database, providing the core functionality of your application.

This controller, combined with the Employee entity and repository, will allow your application to manage employee data in the database using Spring Boot's powerful MVC and JPA features.

### Step 6: Create the EmployeeRepository Interface

Now, let's create the EmployeeRepository interface that will interact with the database and manage Employee entities. This repository will extend the JpaRepository interface provided by Spring Data JPA, which simplifies CRUD operations.

1. Create the EmployeeRepository Interface:  
In the com.example.employeecrud.repository package, create a new interface named EmployeeRepository.java.

```
package com.example.employeecrud.repository;

import com.example.employeecrud.model.Employee;
import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository extends JpaRepository<Employee, Integer>
{
}
```

### Explanation of Each Part:

1. @Repository(Implicit):  
The JpaRepository interface is automatically annotated with @Repository, which means it will be recognized as a Spring Data repository. This allows Spring to handle database operations like saving, updating, and deleting entities without the need for custom queries.
2. JpaRepository<Employee,Integer>:  
The JpaRepository provides several built-in methods to interact with the database, such as save(), findById(), findAll(), deleteById(), and more.
  - Employee is the entity type that this repository will manage.

- Integer is the type of the primary key (id) of the Employee entity. By specifying Integer, we are telling Spring Data JPA that the id field is of type Integer.
3. NoCustomMethodsNeeded:
- Since JpaRepository already provides basic CRUD operations, you don't need to write any custom methods to perform common database tasks (e.g., saving, finding, and deleting employees). However, you can add custom queries later if necessary by defining method signatures in this interface.

### Why Add This Repository?

- DataAccessLayer:  
The repository acts as a bridge between the database and your application. It handles data retrieval and persistence, allowing you to focus on business logic instead of SQL queries.
- SpringDataJPA:  
By extending JpaRepository, you gain access to a wide range of pre-built methods for interacting with the database. These methods make CRUD operations easy to perform without writing SQL queries.
- DecouplesBusinessLogic:  
The repository abstracts away the details of database interaction, so the controller can simply call the repository methods to manage employee data, promoting separation of concerns in the application.

By using this EmployeeRepository, your controller will be able to perform CRUD operations without needing to manually write database interaction code. It will also simplify testing and maintenance by centralizing data access logic.

### Step 7: Add HTML Files in templates Folder

Now, let's create the three HTML files that will serve as the front-end views for the CRUD operations. These files will be placed in the src/main/resources/templates folder and will be rendered by Thymeleaf.

## AddaddEmployee.html

This page will display the form to add a new employee to the system. It uses the `th:action` and `th:field` attributes to bind form data to the `Employee` object, allowing for dynamic form submission and data binding.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Add Employee</title>
</head>
<body>
<h2>Add New Employee</h2>
<form th:action="@{/addEmployee}" th:object="${employee}" method="post">
  <label for="name">Name:</label>
  <input type="text" id="name" th:field="*{name}" placeholder="Enter Name"
required><br><br>

  <label for="email">Email:</label>
  <input type="email" id="email" th:field="*{email}" placeholder="Enter
Email" required><br><br>

  <label for="password">Password:</label>
  <input type="password" id="password" th:field="*{password}"
placeholder="Enter Password" required><br><br>

  <button type="submit">Add Employee</button>
</form>
<a href="/employees">Back to Employee List</a>
</body>
</html>
```

## AddeditEmployee.html

This page will allow users to edit the details of an existing employee. It uses a form with pre-filled data (retrieved from the `Employee` object passed from the controller) and submits the updated details to the server.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Edit Employee</title>
```

```

</head>
<body>
<h2>Edit Employee</h2>
<form th:action="@{/editEmployee/{id}(id=${employee.id})}"
th:object="${employee}" method="post">
  <label for="id">ID:</label>
  <input type="text" id="id" th:field="*{id}" readonly><br><br>

  <label for="name">Name:</label>
  <input type="text" id="name" th:field="*{name}" required><br><br>

  <label for="email">Email:</label>
  <input type="email" id="email" th:field="*{email}" required><br><br>

  <label for="password">Password:</label>
  <input type="password" id="password" th:field="*{password}"
required><br><br>

  <button type="submit">Save Changes</button>
</form>
<a href="/employees">Back to Employee List</a>
</body>
</html>

```

### AddemployeeList.html

This page will display a table listing all employees in the database. Each row includes options to edit or delete the employee, linking to the respective actions.

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Employee List</title>
</head>
<body>
<h2>Employee List</h2>
<table border="1">
  <thead>
    <tr>
      <th>ID</th>
      <th>Name</th>
      <th>Email</th>
      <th>Password</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="employee : ${employees}">
      <td th:text="${employee.id}"></td>
      <td th:text="${employee.name}"></td>
      <td th:text="${employee.email}"></td>

```

```

        <td th:text="{employee.password}"></td>
        <td>
            <a th:href="@{/editEmployee/{id} (id={employee.id})}">Edit</a> |
            <a th:href="@{/deleteEmployee/{id} (id={employee.id})}" onclick="return
confirm('Are you sure you want to delete this employee?')">Delete</a>
        </td>
    </tr>
</tbody>
</table>
<br>
<a href="/addEmployee">Add New Employee</a>
</body>
</html>

```

### Explanation of the HTML Files:

These HTML files serve as the user interface for the application and perform the following logical tasks:

1. **addEmployee.html:**

This page provides a form for the user to input the details of a new employee. It uses the Thymeleaf syntax (th:field) to bind the form fields to the Employee model object. When the form is submitted, it triggers a POST request to the /addEmployee endpoint in the controller.

2. **editEmployee.html:**

This page is used for editing an existing employee's information. The form is pre-filled with the employee's current data (retrieved from the database). The form submits the updated data to the server, where the EmployeeController will update the record in the database.

3. **employeeList.html:**

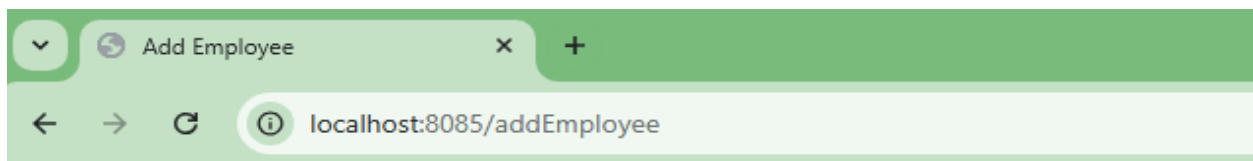
This page displays a table of all employees, showing their ID, name, email, and password. For each employee, there are options to either edit or delete their details. The edit and delete actions trigger the corresponding methods in the EmployeeController.

These pages are connected to the backend via Thymeleaf and Spring MVC, enabling the application to handle CRUD operations seamlessly. Each page corresponds to a specific part of the CRUD process, ensuring that users can easily manage employee data.

### Step 8: Run the Application

Access using

`http://localhost:8085/addEmployee`



## Add New Employee

Name:

Email:

Password:

[Back to Employee List](#)

Add more users

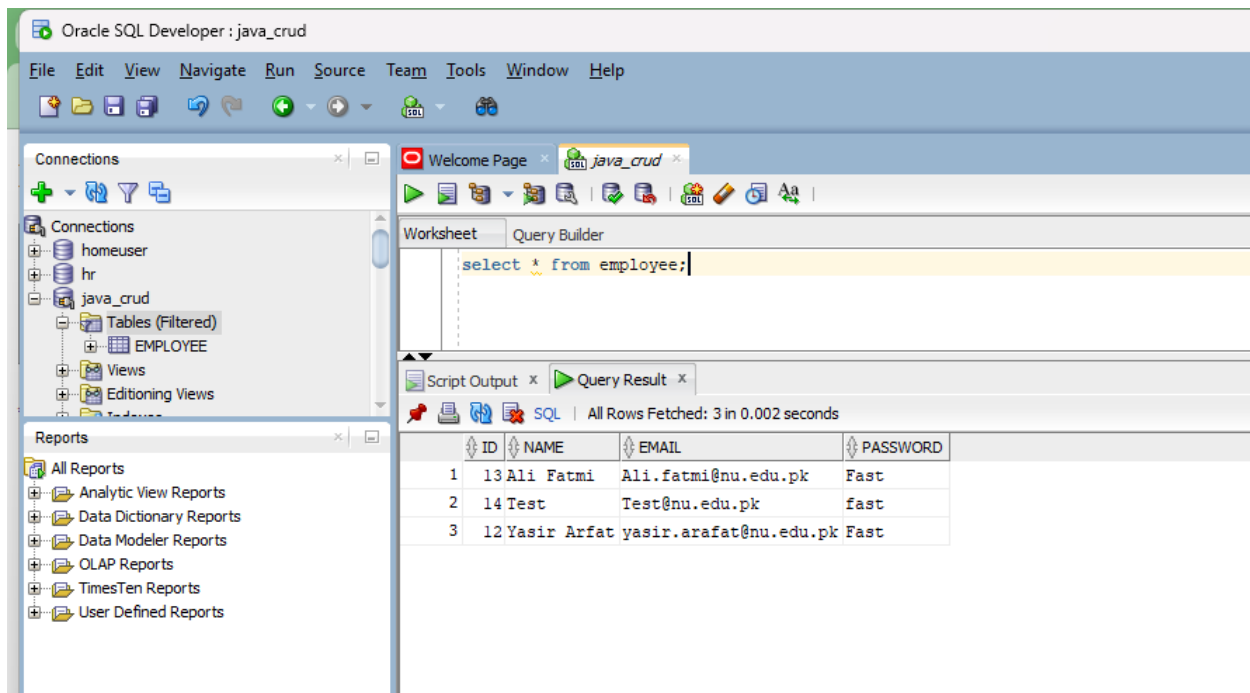


## Employee List

ID	Name	Email	Password	Actions
13	Ali Fatmi	Ali.fatmi@nu.edu.pk	Fast	<a href="#">Edit</a>   <a href="#">Delete</a>
14	Test	Test@nu.edu.pk	fast	<a href="#">Edit</a>   <a href="#">Delete</a>
12	Yasir Arfat	yasir.arafat@nu.edu.pk	Fast	<a href="#">Edit</a>   <a href="#">Delete</a>

[Add New Employee](#)

Verify from database



The screenshot displays the Oracle SQL Developer interface. The 'Connections' pane on the left shows the 'java\_crud' connection selected, with the 'EMPLOYEE' table listed under 'Tables (Filtered)'. The 'Script Output' pane at the bottom shows the query result for the SQL statement 'select \* from employee;'. The result is a table with 3 rows and 4 columns: ID, NAME, EMAIL, and PASSWORD.

ID	NAME	EMAIL	PASSWORD
13	Ali Fatmi	Ali.fatmi@nu.edu.pk	Fast
14	Test	Test@nu.edu.pk	fast
12	Yasir Arfat	yasir.arafat@nu.edu.pk	Fast

Edit Uer



## Edit Employee

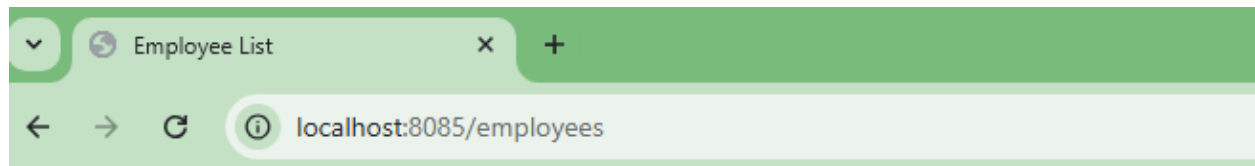
ID:

Name:

Email:

Password:

[Back to Employee List](#)

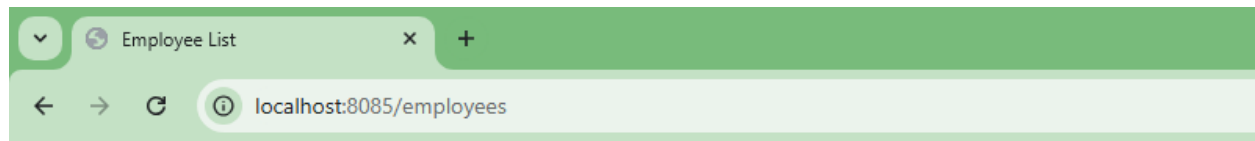


## Employee List

ID	Name	Email	Password	Actions
13	Ali Fatmi	Ali.fatmi@nu.edu.pk	Fast	<a href="#">Edit</a>   <a href="#">Delete</a>
14	Test Data	Test@nu.edu.pk	Fast	<a href="#">Edit</a>   <a href="#">Delete</a>
12	Yasir Arfat	yasir.arafat@nu.edu.pk	Fast	<a href="#">Edit</a>   <a href="#">Delete</a>

[Add New Employee](#)

Delete User



## Employee List

ID	Name	Email	Password	Actions
13	Ali Fatmi	Ali.fatmi@nu.edu.pk	Fast	<a href="#">Edit</a>   <a href="#">Delete</a>
12	Yasir Arfat	yasir.arafat@nu.edu.pk	Fast	<a href="#">Edit</a>   <a href="#">Delete</a>

[Add New Employee](#)

## Session and Cookies in Spring Boot

In Spring Boot, **sessions** and **cookies** are mechanisms to manage user data during interactions with a web application. They are key components for maintaining state in a stateless protocol like HTTP.

### 1. Session

A **session** is a server-side mechanism that tracks user-specific data throughout the user's interaction with a web application.

**How it works:**

#### 1. Session Creation:

- When a user logs in or interacts with the application, a session is created on the server.
- Spring Boot assigns a unique session ID to the user.

#### 2. Session Storage:

- Data related to the session is stored on the server (e.g., in memory, database, or Redis).
- The session ID is sent to the user in a cookie or URL parameter.

#### 3. Session Lifecycle:

- The session persists until the user logs out, the session times out, or the application invalidates it.

**Common Use Cases:**

- Storing user authentication details (e.g., logged-in status).
- Tracking shopping cart items in an e-commerce application.

**Spring Boot Implementation:**

Spring Boot uses `HttpSession` to manage sessions. Example:

```
import org.springframework.web.bind.annotation.*;  
  
import javax.servlet.http.HttpSession;
```

```
@RestController

public class SessionController {

    @GetMapping("/setSession")
    public String setSession(HttpSession session) {
        session.setAttribute("username", "Ali");
        return "Session data set";
    }

    @GetMapping("/getSession")
    public String getSession(HttpSession session) {
        return "Hello, " + session.getAttribute("username");
    }
}
```

## 2. Cookies

A **cookie** is a small piece of data stored on the client-side (browser). It is sent back to the server with each request.

**How it works:**

### 1. Setting a Cookie:

- The server sends a Set-Cookie header in the HTTP response.
- The browser stores the cookie.

### 2. Sending Cookies to the Server:

- The browser includes cookies in subsequent requests to the server.

### 3. Cookie Expiry:

- Cookies can be persistent (with an expiration date) or session cookies (cleared when the browser is closed).

### Common Use Cases:

- Tracking user preferences (e.g., language).
- Managing lightweight data like a session ID.

### Spring Boot Implementation:

Spring Boot uses the `javax.servlet.http.Cookie` class. Example:

```
import org.springframework.web.bind.annotation.*;
```

```
import javax.servlet.http.Cookie;
```

```
import javax.servlet.http.HttpServletResponse;
```

```
import javax.servlet.http.HttpServletRequest;
```

```
@RestController
```

```
public class CookieController {
```

```
    @GetMapping("/setCookie")
```

```
    public String setCookie(HttpServletResponse response) {
```

```
        Cookie cookie = new Cookie("username", "Ali");
```

```
        cookie.setMaxAge(7 * 24 * 60 * 60); // 7 days
```

```
        response.addCookie(cookie);
```

```
        return "Cookie set";
```

```
    }
```

```
    @GetMapping("/getCookie")
```

```
    public String getCookie(@CookieValue(name = "username", defaultValue = "Guest") String  
username) {
```

```
        return "Hello, " + username;
```

```
    }
```

```
}
```

## Key Differences Between Session and Cookies

Aspect	Session	Cookies
Storage Location	Server	Client (browser)
Data Size	Can store large data	Limited to ~4KB
Security	More secure (server-side)	Less secure (stored on client)
Persistence	Ends with session timeout	Can persist beyond sessions

## Integration in Spring Boot Applications

- **Cookies** are often used to store session IDs.
- **Sessions** are used for maintaining state, especially for sensitive or complex data.

Both can be configured in application.properties for Spring Boot:

`server.servlet.session.timeout=30m` # Session timeout

`server.servlet.session.cookie.name=MYSESSIONID` # Cookie name for session

This combination ensures a seamless and secure experience for users in web applications.

## User Login with Session and Cookies

Now, we will add functionality for user login to our project above using sessions and cookies. This will ensure that after a user logs in, their session persists, and their user ID is stored in a cookie for easier management.

### 1. Create UserController

Add the following UserController to your project under the com.example.employeecrud.controller package.

```
package com.example.employeecrud.controller;

import com.example.employeecrud.model.Employee;
import com.example.employeecrud.service.UserService;
import jakarta.servlet.http.Cookie;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import org.springframework.beans.factory.annotation.Autowired;
```

```

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

@Controller
@RequestMapping("/user")
public class UserController {
    @Autowired
    private UserService userService;

    // Login page
    @RequestMapping(value = {"/", ""})
    public String loginPage() {
        return "login";
    }

    // Handle user login
    @PostMapping("/")
    public String loginUser(@RequestParam String email, @RequestParam String
password,
                            HttpSession session, HttpServletResponse
response, Model model) {
        Employee user = userService.validateUser(email, password);
        if (user != null) {
            session.setAttribute("user", user); // Store user in session
            Cookie cookie = new Cookie("userId",
String.valueOf(user.getId()));
            response.addCookie(cookie); // Add cookie
            return "redirect:/user/dashboard"; // Redirect to dashboard
        } else {
            model.addAttribute("error", "Invalid email or password");
            return "login";
        }
    }

    // Dashboard (accessible only if logged in)
    @GetMapping("/dashboard")
    public String dashboard(HttpSession session, Model model) {
        Employee user = (Employee) session.getAttribute("user");
        if (user == null) {
            return "redirect:/user/"; // Redirect to login if not logged in
        }
        model.addAttribute("user", user); // Add user details to model
        return "dashboard";
    }

    // Logout functionality
    @GetMapping("/logout")
    public String logout(HttpSession session, HttpServletResponse response) {
        session.invalidate(); // Clear session
        Cookie cookie = new Cookie("userId", null);
        cookie.setMaxAge(0);
        response.addCookie(cookie); // Remove cookie
        return "redirect:/user/";
    }
}

```



## Explanation

- **Login:**
  - The loginPage method renders the login page (login.html).
  - The loginUser method validates the user's email and password using the UserService. If valid:
    - The user is stored in the session.
    - A cookie with the user ID is added to the browser.
    - The user is redirected to the dashboard.
- **Dashboard:**
  - The dashboard method ensures the user is logged in by checking the session. If no user is found in the session, it redirects to the login page. Otherwise, it displays the dashboard.
- **Logout:**
  - The logout method clears the session and removes the cookie, then redirects to the login page.

## Add UserRepository

Now, we need to add a repository interface for user-related operations. This repository will allow us to query the database for user details.

## Code

Add the following UserRepository to the com.example.employeecrud.repository package:

```
package com.example.employeecrud.repository;

import com.example.employeecrud.model.Employee;
import com.example.employeecrud.model.Employee;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository
public interface UserRepository extends JpaRepository<Employee, Long> {
    Optional<Employee> findByEmail(String email);
}
```

## Explanation

- **JpaRepository<Employee, Long>**: This defines that UserRepository is a JPA repository for the Employee entity with the primary key of type Long.
- **findByEmail(String email)**: This method is used to find an Employee by their email. It returns an Optional<Employee> for safe handling of null values.

This repository will be used in the UserService for validating the user's email and password.

## Add UserService

Now, create a service class for handling user validation logic. This service will interact with the repository layer to retrieve and validate user credentials.

## Code

Add the following UserService to a new package named com.example.employeecrud.service:

```
package com.example.employeecrud.service;

import com.example.employeecrud.model.Employee;
import com.example.employeecrud.repository.EmployeeRepository;
import com.example.employeecrud.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Optional;

@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public Employee validateUser(String email, String password) {
        Optional<Employee> user = userRepository.findByEmail(email);
        return user.filter(u ->
            u.getPassword().equals(password)).orElse(null);
    }
}
```

## Explanation

- **@Service:** Marks this class as a service layer component in the Spring application context.
- **UserRepository:** Injected using @Autowired, this repository is used to fetch user data from the database.
- **validateUser(String email, String password):**
  - Retrieves a user by their email using findByEmail.
  - Uses filter to check if the password matches the provided one.
  - Returns the user object if the credentials are valid or null if not.

This service ensures the separation of business logic (validation) from the controller and repository layers.

## Add HTML Templates for User Authentication

To handle the login and user dashboard functionalities, add the following HTML files in the src/main/resources/templates folder.

### Login.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Login</title>
</head>
<body>
<h2>Login</h2>
<form action="/user/" method="post">
  <label>Email:</label>
  <input type="email" name="email" required>
  <br>
  <label>Password:</label>
  <input type="password" name="password" required>
  <br>
  <button type="submit">Login</button>
</form>
<p style="color:red;">${error}</p>
</body>
</html>
```

## Dashboard.html

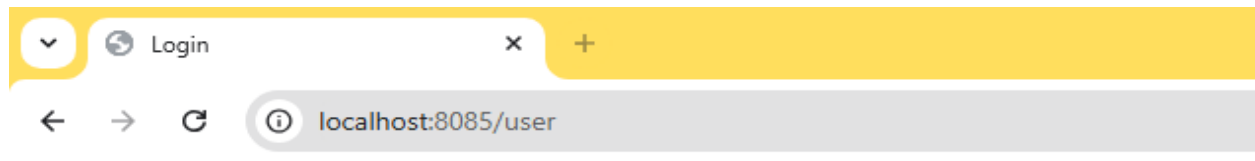
```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>User Dashboard</title>
</head>
<body>
<h2>Welcome, <span th:text="${user.name}">[User Name]</span>!</h2>
<p><strong>Email:</strong> <span th:text="${user.email}">[User
Email]</span></p>
<p><strong>User ID:</strong> <span th:text="${user.id}">[User ID]</span></p>

<!-- Correct logout URL -->
<form action="/user/logout" method="get">
  <button type="submit">Logout</button>
</form>
</body>
</html>
```

### Explanation:

- Displays the logged-in user's details, such as their name, email, and user ID, using Thymeleaf variables (`${user.name}`, etc.).
- The logout button sends a GET request to `/user/logout` to log out the user by clearing the session and cookies.

### Run Application

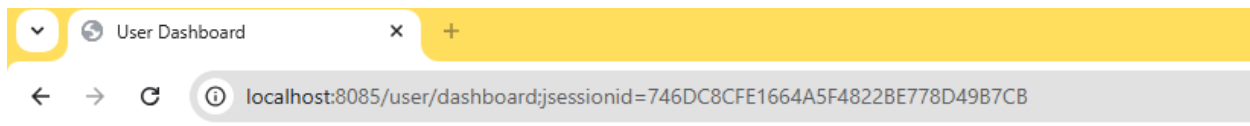


## Login

Email:

Password:

\${error}



## Welcome, Yasir Arfat!

Email: yasir.arafat@nu.edu.pk

User ID: 12

## THE END OF SCD LAB

This is the end of the SCD Lab. I hope you have gained valuable skills and insights throughout this course. Best of luck in your future endeavors, and may your learning journey continue to thrive.