

INFORMATION SYSTEM ANALYSIS & DESIGN FINAL EXAMINATION

Updated date: 03.12.2021

TIỂU LUẬN & VẤN ĐÁP

Sinh viên viết thành tài liệu (Part 1 & 2)

//tiếng Việt, Anh hay kết hợp đều được

//Show cho giảng viên khi thi

//và trả lời câu hỏi liên quan

The more you know, the more you understand you know nothing

Part 1: Question

Q1: Styles of software architecture

<https://www.tutorialride.com/software-engineering/architectural-styles-for-software-design.htm>

<https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013>

Q2: Steps for analysis

https://www.tutorialspoint.com/system_analysis_and_design/system_analysis_and_design_quick_guide.htm

Q3: Steps for design (cohesion, coupling....give example, Relationship, code...)

https://www.tutorialspoint.com/system_analysis_and_design/system_analysis_and_design_quick_guide.htm

<https://www.javatpoint.com/software-engineering-coupling-and-cohesion>

<https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/>

Q4: 3 Layer architecture style & MVC

<https://crimsonpublishers.com/prsp/fulltext/PRSP.000505.php>

<https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures>

Q5: DAO? Why? class diagram & code for DAO. Review Appendix B1 & Explain diagram

<https://colin-but.medium.com/dao-pattern-explained-895b65436f1c>

Q6: Database design

<https://www.lucidchart.com/pages/database-diagram/database-design>

Q7: Interface design (principles...)

https://www.tutorialspoint.com/software_engineering/software_user_interface_design.htm

<https://www.interaction-design.org/literature/topics/ui-design>

<https://www.usability.gov/what-and-why/user-interface-design.html>

<https://xd.adobe.com/ideas/process/ui-design/4-golden-rules-ui-design/>

Part 2: Project

Write a document of the system analysis and design: **Book store online**

- Collect requirements: Students collect information from webs such as online stores tiki, amazon, eBay...for more functionality

- Perform the system analysis and design
 - Implement some functions (utilize **jsp servlet**)
 - Customer (register, login)
 - Customer creates order (crawler, search, add to cart, shipment, payment)
 - Staff (login, store book-information, update & push book-information on website)
1. Draw a use case diagram with detail level
 2. Construct a class diagram in analysis (**NO** 1...m, *ORM*) with some main methods
 3. Construct a class diagram for data model (**ADD** 1...m, *ORM*), *generate* data model and Database mySQL
 4. Construct a class diagram in design (**ADD** more attributes and methods).
 5. Using the pattern DAO, redesign classes in your system
 6. Construct package diagrams with 3-layer architecture with MVC for your systems (follow the pattern given by tdque@yahoo.com).
 7. Implementation (**utilize jsp servlet**)

References

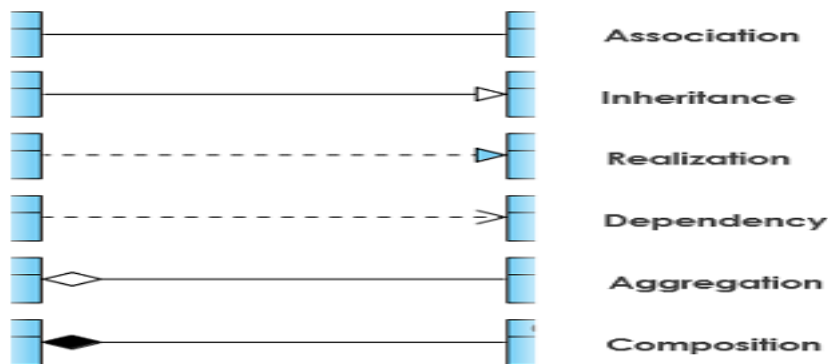
- [1] JSP Servlet JDBC MySQL CRUD. Available at:
<https://www.javaguides.net/2019/03/jsp-servlet-jdbc-mysql-crud-example-tutorial.html>
<https://www.javaguides.net/2019/03/registration-form-using-jsp-servlet-jdbc-mysql-example.html>
<https://www.javaguides.net/2019/03/login-form-using-jsp-servlet-jdbc-mysql-example.html>
- [2] <https://www.codejava.net/coding/jsp-servlet-jdbc-mysql-create-read-update-delete-crud-example>
- [3] <https://www.visual-paradigm.com/download/>
<https://www.visual-paradigm.com/tutorials>
<https://www.visual-paradigm.com/guides>
- [4] UML organization web <https://www.uml-diagrams.org/>
- [5] DFD model
<https://www.visual-paradigm.com/tutorials/data-flow-diagram-example-food-ordering-system.jsp>
- [6] ACTIVITY-SWIMLANE
<https://online.visual-paradigm.com/diagrams/features/flowchart-tool/swimlane-diagram-tool/>
- [7] STATE DIAGRAM
<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-state-machine-diagram/>
- [8] ENTITY RELATIONSHIP DIAGRAM
<https://www.visual-paradigm.com/guide/data-modeling/what-is-entity-relationship-diagram/>
- [9] User Stories & acceptance criteria
<https://www.altexsoft.com/blog/business/acceptance-criteria-purposes-formats-and-best-practices/>
- [10] data model: <https://online.visual-paradigm.com/knowledge/visual-modeling/conceptual-vs-logical-vs-physical-data-model/>

APPENDIX A

UML Class Diagram and Relationships

Question:

- How can we determine classes while developing a software system?
 - **When:** in which phase in software process?
 - **Where:** Class name/attributes/methods will define from where?
 - **Who:** who takes part in these activities?
- How many relationships among classes in UML are there?
- Given two classes A, B, how can we define a relationship between these two classes?
- Is there more than one relationship between two classes? If more than one, which one should be selected? Selection depends what?
- Class relationship depends on **business process**?
- Using Design pattern DAO (Data Access Object) why?



There are six relationships between classes in a UML class diagram :

Dependency

Association

Aggregation

Composition

Inheritance

Realization

The above relationships are read as follows:

- Dependency : class A uses class B
- Association: class A associates with class B
- Aggregation : class A has a class B
- Composition : class A owns a class B
- Inheritance : class B is a Class A (or class A is extended by class B)
- Realization : class B realizes Class A (or class A is realized by class B)

Dependency/association is represented when a reference to one class is passed in as a method parameter to another class. For example, an instance of class B is passed in to a method of class A:

```
public class A {  
    public void doSomething(B b) { }  
}
```

Aggregation: If class A stored the reference to class B for later use we would have a different relationship called Aggregation. A more common and more obvious example of Aggregation would be via setter injection:

```
public class A {  
    private B bb;  
    public void setB(B b) { bb = b; }  
}
```

Composition: Aggregation is the weaker form of object containment (one object contains other objects). The stronger form is called Composition. In Composition the containing object is responsible for the creation and life cycle of the contained object. Implementation of composition is of the following forms:

- First, via member initialization:

```
public class A {  
    private B b = new B();  
}
```

- Second, via constructor initialization:

```
public class A {  
    private B bb;  
    public A() {  
        bb = new B();  
    }  
}
```

// default constructor

- Third, via lazy init:

```
public class A {  
    private B _b;  
    public B getB() {  
        if (null == _b) {  
            _b = new B();  
        }  
        return _b;  
    }  
} // getB()
```

Inheritance is a fairly straightforward relationship to depict in Java:

```
public class A {  
    ...  
} // class A
```

```
public class B extends A {  
    ....  
} // class B
```

Realization is also straightforward in Java and deals with implementing an interface:

```
public interface A {  
    ...  
} // interface A  
public class B implements A { ....  
} // class B
```

APPENDIX B1

JSP Servlet JDBC MySQL CRUD

By Ramesh Fadatare

<https://www.javaguides.net/2019/03/jsp-servlet-jdbc-mysql-crud-example-tutorial.html>

In this tutorial, we are building a simple **User Management** web application that manages a collection of **users** with the basic feature:

list, insert, update, delete (or CRUD operations - Create, Update, Read and Delete).

You can download the source code of this tutorial from my GitHub repository and the link is given at the end of this tutorial.

Top JSP, Servlet and JDBC Tutorials:

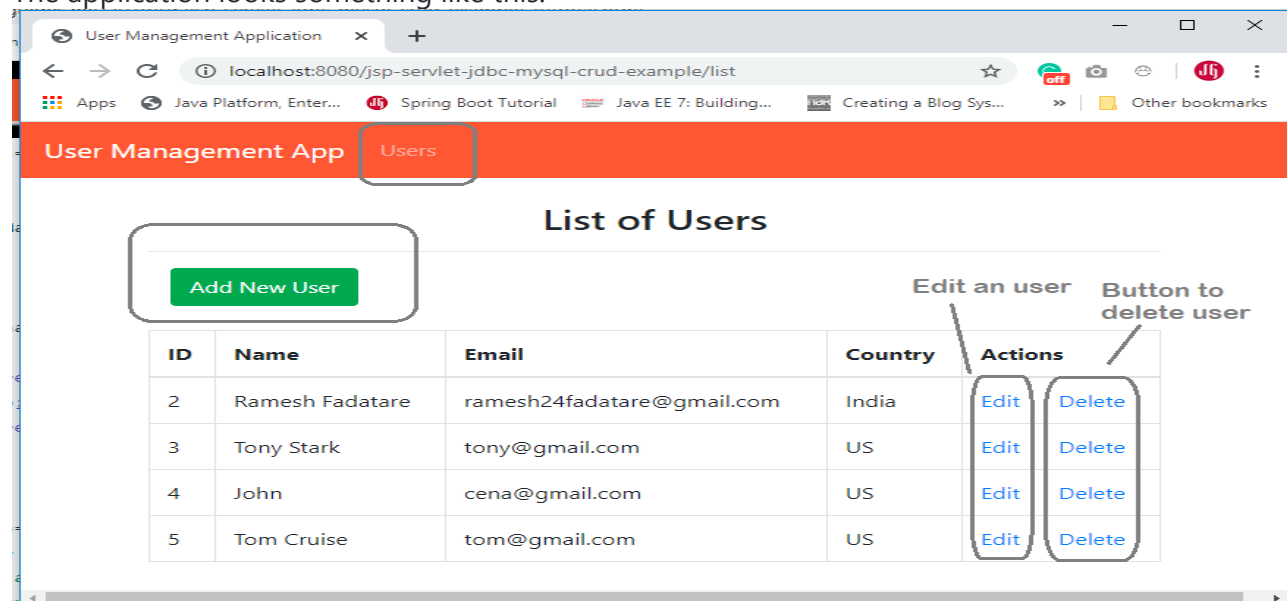
- [Servlet Tutorial](#)
- [JSP Tutorial](#)
- [JDBC 4.2 Tutorial](#)

Check out [Build Todo App using JSP, Servlet, JDBC, and MySQL](#).

We will develop below simple basic features in our **User Management** web application:

1. Create a User
2. Update a User
3. Delete a User
4. Retrieve a User
5. List of all Users

The application looks something like this:



Tools and technologies used

- JSP - 2.2 +
- IDE - STS/Eclipse Neon.3
- JDK - 1.8 or later

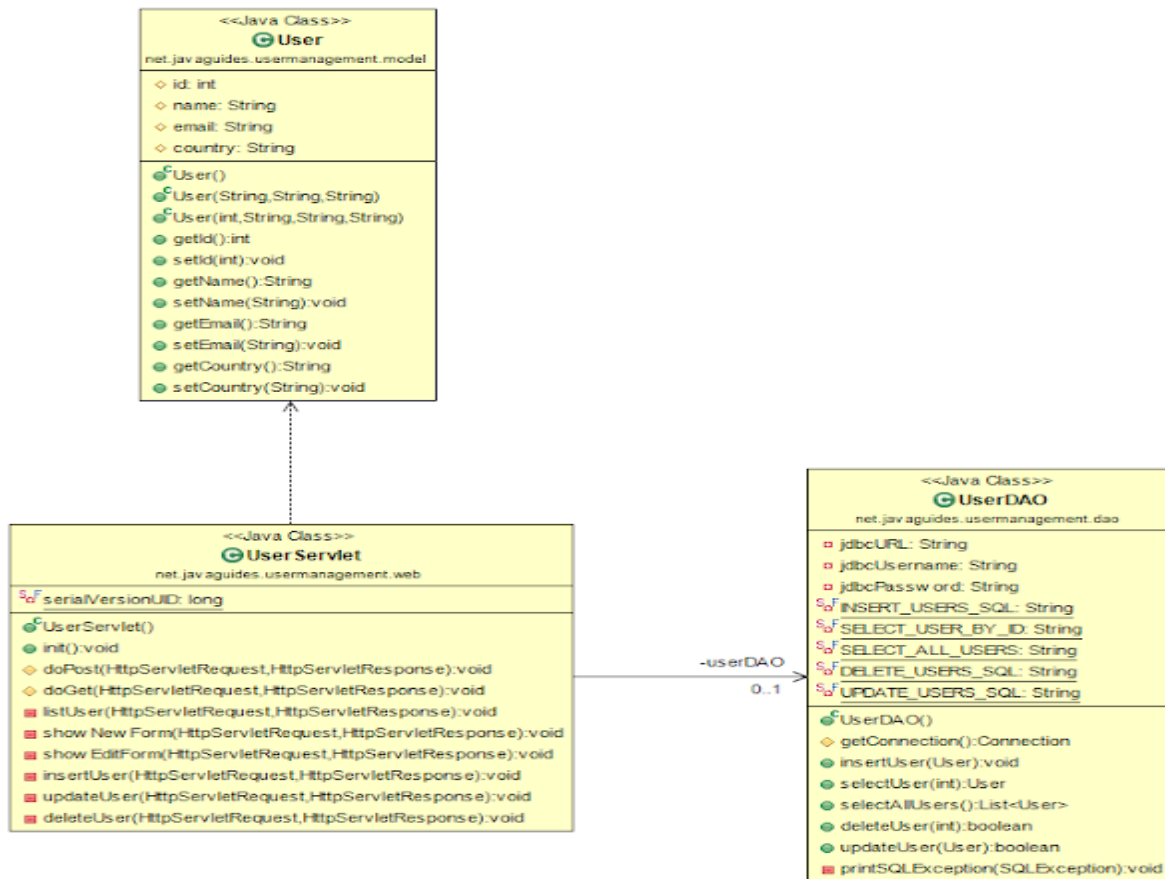
- Apache Tomcat - 8.5
- JSTL - 1.2.1
- **Servlet API** - 2.5
- MySQL - mysql-connector-java-8.0.13.jar

Development Steps

1. Create an Eclipse Dynamic Web Project
2. Add Dependencies
3. Project Structure
4. MySQL Database Setup
5. Create a JavaBean - User.java
6. Create a UserDao.java
7. Create a UserService.java
8. Creating User Listing JSP Page - user-list.jsp
9. Create a User Form JSP Page - user-form.jsp
10. Creating Error JSP page
11. Deploying and Testing the Application Demo

Class Diagram

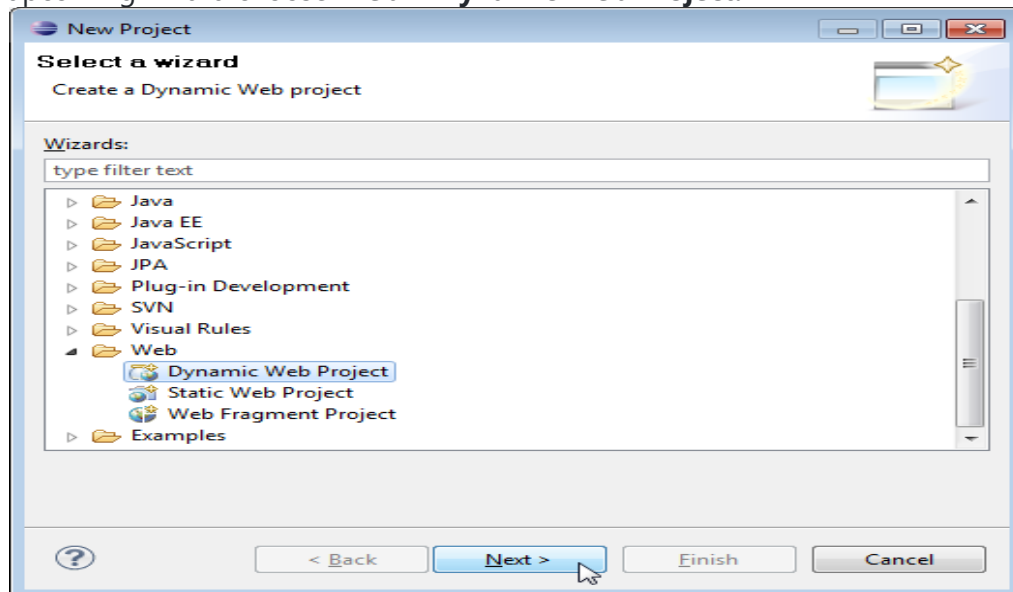
Here is the class diagram of the **User Management** web application that we are going to develop in this tutorial:



1. Create an Eclipse Dynamic Web Project

To create a new dynamic Web project in Eclipse:

1. On the main menu select **File > New > Project....**
2. In the upcoming wizard choose **Web > Dynamic Web Project**.



3. Click **Next**.
4. Enter project name as "jsp-servlet-jdbc-mysql-example";
5. Make sure that the target runtime is set to Apache Tomcat with the currently supported version.

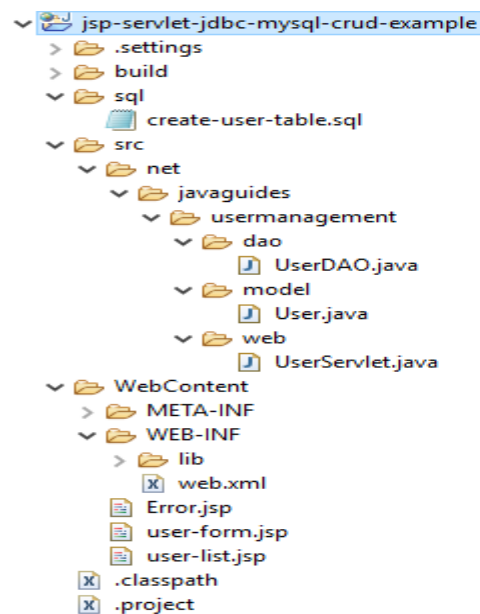
2. Add Dependencies

Add the latest release of below jar files to the **lib** folder.

- jsp-api.2.3.1.jar
- servlet-api.2.3.jar
- mysql-connector-java-8.0.13.jar
- jstl-1.2.jar

3. Project Structure

Standard project structure for your reference -



4. MySQL Database Setup

Let's create a database named "demo" in MySQL. Now, create a users table using below DDL script:

```
CREATE DATABASE 'demo';
USE demo;

create table users (
  id int(3) NOT NULL AUTO_INCREMENT,
  name varchar(120) NOT NULL,
```



```
email varchar(220) NOT NULL,  
country varchar(120),  
PRIMARY KEY (id)  
);
```

You can use either MySQL Command Line Client or MySQL Workbench tool to create the database. The above a **users** table looks like:

```
mysql> desc users;  
+-----+-----+-----+-----+-----+-----+  
| Field | Type          | Null | Key | Default | Extra          |  
+-----+-----+-----+-----+-----+-----+  
| id    | int(3)        | NO   | PRI | NULL    | auto_increment |  
| name  | varchar(120)  | NO   |     | NULL    |                |  
| email | varchar(220)  | NO   |     | NULL    |                |  
| country | varchar(120) | YES  |     | NULL    |                |  
+-----+-----+-----+-----+-----+-----+  
4 rows in set (0.00 sec)
```

5. Create a JavaBean - User.java

Let's create a *User* java class to model a **user** entity in the database with the following code:

```
package net.javaguides.usermanagement.model;  
  
/**  
 * User.java  
 * This is a model class represents a User entity  
 * @author Ramesh Fadatara  
 */  
public class User {  
    protected int id;  
    protected String name;  
    protected String email;  
    protected String country;  
  
    public User() {}  
  
    public User(String name, String email, String country) {  
        super();  
        this.name = name;  
        this.email = email;  
        this.country = country;  
    }  
  
    public User(int id, String name, String email, String country) {  
        super();  
        this.id = id;  
        this.name = name;  
        this.email = email;  
        this.country = country;  
    }  
}
```

```

    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 getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
}

```

6. Create a UserDAO.java

Let's create a *UserDAO* class which is a Data Access Layer (DAO) class that provides **CRUD** (**Create, Read, Update, Delete**) operations for the table **users** in a database. Here's the full source code of the *UserDAO*:

```

package net.javaguides.usermanagement.dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import net.javaguides.usermanagement.model.User;

/**
 * AbstractDAO.java This DAO class provides CRUD database operations for the
 * table users in the database.
 *
 * @author Ramesh Fadatara

```

```

*
*/
public class UserDAO {
    private String jdbcURL = "jdbc:mysql://localhost:3306/demo?useSSL=false";
    private String jdbcUsername = "root";
    private String jdbcPassword = "root";

    private static final String INSERT_USERS_SQL = "INSERT INTO users" + " (name, email, country) VALUES " + " (?, ?, ?);";

    private static final String SELECT_USER_BY_ID = "select id,name,email,country from users where id =?";
    private static final String SELECT_ALL_USERS = "select * from users";
    private static final String DELETE_USERS_SQL = "delete from users where id = ?;";
    private static final String UPDATE_USERS_SQL = "update users set name = ?,email= ?, country =? where id = ?;";

    public UserDAO() {}

    protected Connection getConnection() {
        Connection connection = null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            connection = DriverManager.getConnection(jdbcURL, jdbcUsername, jdbcPassword);
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return connection;
    }

    public void insertUser(User user) throws SQLException {
        System.out.println(INSERT_USERS_SQL);
        // try-with-resource statement will auto close the connection.
        try (Connection connection = getConnection(); PreparedStatement preparedStatement = connection.prepareStatement(INSERT_USERS_SQL)) {
            preparedStatement.setString(1, user.getName());
            preparedStatement.setString(2, user.getEmail());
            preparedStatement.setString(3, user.getCountry());
            System.out.println(preparedStatement);
            preparedStatement.executeUpdate();
        } catch (SQLException e) {
            printSQLException(e);
        }
    }

    public User selectUser(int id) {
        User user = null;
        // Step 1: Establishing a Connection
        try (Connection connection = getConnection();
            // Step 2: Create a statement using connection object

```

```

        PreparedStatement preparedStatement =
connection.prepareStatement(SELECT_USER_BY_ID);) {
    preparedStatement.setInt(1, id);
    System.out.println(preparedStatement);
    // Step 3: Execute the query or update query
    ResultSet rs = preparedStatement.executeQuery();

    // Step 4: Process the ResultSet object.
    while (rs.next()) {
        String name = rs.getString("name");
        String email = rs.getString("email");
        String country = rs.getString("country");
        user = new User(id, name, email, country);
    }
} catch (SQLException e) {
    printSQLException(e);
}
return user;
}

public List < User > selectAllUsers() {

    // using try-with-resources to avoid closing resources (boiler plate code)
    List < User > users = new ArrayList < > ();
    // Step 1: Establishing a Connection
    try (Connection connection = getConnection());

        // Step 2: Create a statement using connection object
        PreparedStatement preparedStatement =
connection.prepareStatement(SELECT_ALL_USERS);) {
        System.out.println(preparedStatement);
        // Step 3: Execute the query or update query
        ResultSet rs = preparedStatement.executeQuery();

        // Step 4: Process the ResultSet object.
        while (rs.next()) {
            int id = rs.getInt("id");
            String name = rs.getString("name");
            String email = rs.getString("email");
            String country = rs.getString("country");
            users.add(new User(id, name, email, country));
        }
    } catch (SQLException e) {
        printSQLException(e);
    }
    return users;
}

public boolean deleteUser(int id) throws SQLException {
    boolean rowDeleted;
    try (Connection connection = getConnection(); PreparedStatement statement =
connection.prepareStatement(DELETE_USERS_SQL);) {
        statement.setInt(1, id);
        rowDeleted = statement.executeUpdate() > 0;
    }
    return rowDeleted;
}

```

```

    }

    public boolean updateUser(User user) throws SQLException {
        boolean rowUpdated;
        try (Connection connection = getConnection(); PreparedStatement statement =
connection.prepareStatement(UPDATE_USERS_SQL);) {
            statement.setString(1, user.getName());
            statement.setString(2, user.getEmail());
            statement.setString(3, user.getCountry());
            statement.setInt(4, user.getId());

            rowUpdated = statement.executeUpdate() > 0;
        }
        return rowUpdated;
    }

    private void printSQLException(SQLException ex) {
        for (Throwable e: ex) {
            if (e instanceof SQLException) {
                e.printStackTrace(System.err);
                System.err.println("SQLState: " + ((SQLException) e).getSQLState());
                System.err.println("Error Code: " + ((SQLException)
e).getErrorCode());
                System.err.println("Message: " + e.getMessage());
                Throwable t = ex.getCause();
                while (t != null) {
                    System.out.println("Cause: " + t);
                    t = t.getCause();
                }
            }
        }
    }
}

```

7. Create a UserServlet.java

Now, let's create *UserServlet* that acts as a page **controller** to handle all requests from the client. Let's look at the code first:

```

package net.javaguides.usermanagement.web;

import java.io.IOException;
import java.sql.SQLException;
import java.util.List;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

```

```

import net.javaguides.usermanagement.dao.UserDAO;
import net.javaguides.usermanagement.model.User;

/**
 * ControllerServlet.java
 * This servlet acts as a page controller for the application, handling all
 * requests from the user.
 * @email Ramesh Fadatare
 */

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

    public void init() {
        userDAO = new UserDAO();
    }

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

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String action = request.getServletPath();

        try {
            switch (action) {
                case "/new":
                    showNewForm(request, response);
                    break;
                case "/insert":
                    insertUser(request, response);
                    break;
                case "/delete":
                    deleteUser(request, response);
                    break;
                case "/edit":
                    showEditForm(request, response);
                    break;
                case "/update":
                    updateUser(request, response);
                    break;
                default:
                    listUser(request, response);
                    break;
            }
        } catch (SQLException ex) {
            throw new ServletException(ex);
        }
    }

    private void listUser(HttpServletRequest request, HttpServletResponse response)
        throws SQLException, IOException, ServletException {

```

```

        List < User > listUser = userDao.selectAllUsers();
        request.setAttribute("listUser", listUser);
        RequestDispatcher dispatcher = request.getRequestDispatcher("user-list.jsp");
        dispatcher.forward(request, response);
    }

    private void showNewForm(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
        RequestDispatcher dispatcher = request.getRequestDispatcher("user-form.jsp");
        dispatcher.forward(request, response);
    }

    private void showEditForm(HttpServletRequest request, HttpServletResponse
response)
    throws SQLException, ServletException, IOException {
        int id = Integer.parseInt(request.getParameter("id"));
        User existingUser = userDao.selectUser(id);
        RequestDispatcher dispatcher = request.getRequestDispatcher("user-form.jsp");
        request.setAttribute("user", existingUser);
        dispatcher.forward(request, response);
    }

    private void insertUser(HttpServletRequest request, HttpServletResponse response)
    throws SQLException, IOException {
        String name = request.getParameter("name");
        String email = request.getParameter("email");
        String country = request.getParameter("country");
        User newUser = new User(name, email, country);
        userDao.insertUser(newUser);
        response.sendRedirect("list");
    }

    private void updateUser(HttpServletRequest request, HttpServletResponse response)
    throws SQLException, IOException {
        int id = Integer.parseInt(request.getParameter("id"));
        String name = request.getParameter("name");
        String email = request.getParameter("email");
        String country = request.getParameter("country");

        User book = new User(id, name, email, country);
        userDao.updateUser(book);
        response.sendRedirect("list");
    }

    private void deleteUser(HttpServletRequest request, HttpServletResponse response)
    throws SQLException, IOException {
        int id = Integer.parseInt(request.getParameter("id"));
        userDao.deleteUser(id);
        response.sendRedirect("list");
    }
}

```

8. Creating User Listing JSP Page - user-list.jsp

Next, create a JSP page for displaying all **users** from the database. Let's create a *list-user.jsp* page under the **WebContent** directory in the project with the following code:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
    <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
    <html>

    <head>
        <title>User Management Application</title>
        <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
        </head>

        <body>

            <header>
                <nav class="navbar navbar-expand-md navbar-dark" style="background-
color: tomato">
                    <div>
                        <a href="https://www.javaguides.net" class="navbar-brand">
User
Management App </a>
                    </div>

                    <ul class="navbar-nav">
                        <li><a href="<%=request.getContextPath()%>/list" class="nav-
link">Users</a></li>
                    </ul>
                </nav>
            </header>
            <br>

            <div class="row">
                <!-- <div class="alert alert-success"
*ngIf='message'>{{message}}</div> -->

                <div class="container">
                    <h3 class="text-center">List of Users</h3>
                    <hr>
                    <div class="container text-left">

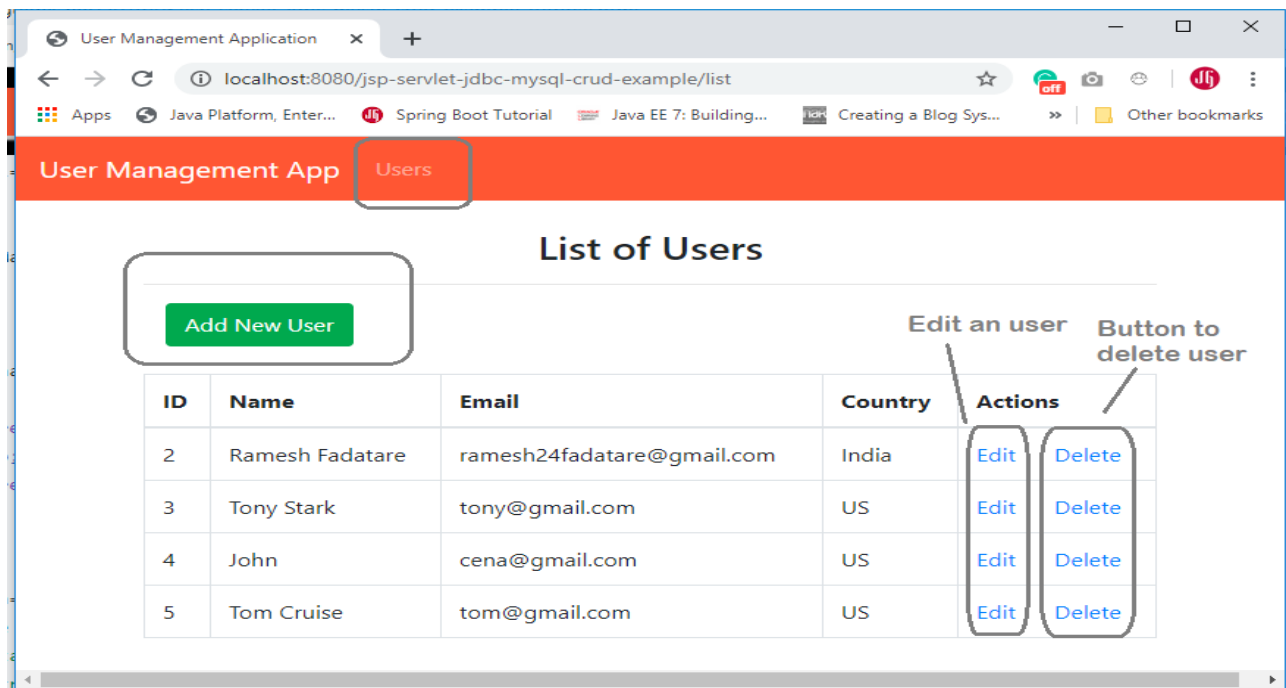
                        <a href="<%=request.getContextPath()%>/new" class="btn btn-
success">Add
New User</a>

                    </div>
                    <br>
                    <table class="table table-bordered">
```



```
<thead>  
    <tr>  
        <th>ID</th>  
        <th>Name</th>  
        <th>Email</th>  
        <th>Country</th>  
        <th>Actions</th>  
    </tr>  
</thead>  
<tbody>  
    <!--      for (Todo todo: todos) { -->  
    <c:forEach var="user" items="${listUser}">  
  
        <tr>  
            <td>  
                <c:out value="${user.id}" />  
            </td>  
            <td>  
                <c:out value="${user.name}" />  
            </td>  
            <td>  
                <c:out value="${user.email}" />  
            </td>  
            <td>  
                <c:out value="${user.country}" />  
            </td>  
            <td><a href="edit?id=<c:out value='${user.id}'  
>">Edit</a>   <a href="delete?id=<c:out value='${user.id}'  
>">Delete</a></td>  
        </tr>  
    </c:forEach>  
    <!-- } -->  
</tbody>  
  
</table>  
</div>  
</div>  
</body>  
  
</html>
```

Once you will deploy above JSP page in tomcat and open in the browser looks something like this:



9. Create a User Form JSP Page - user-form.jsp

Next, we create a JSP page for creating a new User called *user-form.jsp*. Here's its full source code:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>

    <head>
        <title>User Management Application</title>
        <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
    </head>

    <body>

        <header>
            <nav class="navbar navbar-expand-md navbar-dark" style="background-
color: tomato">
                <div>
                    <a href="https://www.javaguides.net" class="navbar-brand">
User Management App </a>
                </div>
            </nav>
        </header>
    </body>
</html>
```

```

        <ul class="navbar-nav">
            <li><a href="<%=request.getContextPath()%>/list" class="nav-
link">Users</a></li>
        </ul>
    </nav>
</header>
<br>
<div class="container col-md-5">
    <div class="card">
        <div class="card-body">
            <c:if test="${user != null}">
                <form action="update" method="post">
            </c:if>
            <c:if test="${user == null}">
                <form action="insert" method="post">
            </c:if>

            <caption>
                <h2>
                    <c:if test="${user != null}">
                        Edit User
                    </c:if>
                    <c:if test="${user == null}">
                        Add New User
                    </c:if>
                </h2>
            </caption>

            <c:if test="${user != null}">
                <input type="hidden" name="id" value="<c:out
value='${user.id}' />" />
            </c:if>

            <fieldset class="form-group">
                <label>User Name</label> <input type="text" value="<c:out
value='${user.name}' />" class="form-control" name="name" required="required">
            </fieldset>

            <fieldset class="form-group">
                <label>User Email</label> <input type="text"
value="<c:out value='${user.email}' />" class="form-control" name="email">
            </fieldset>

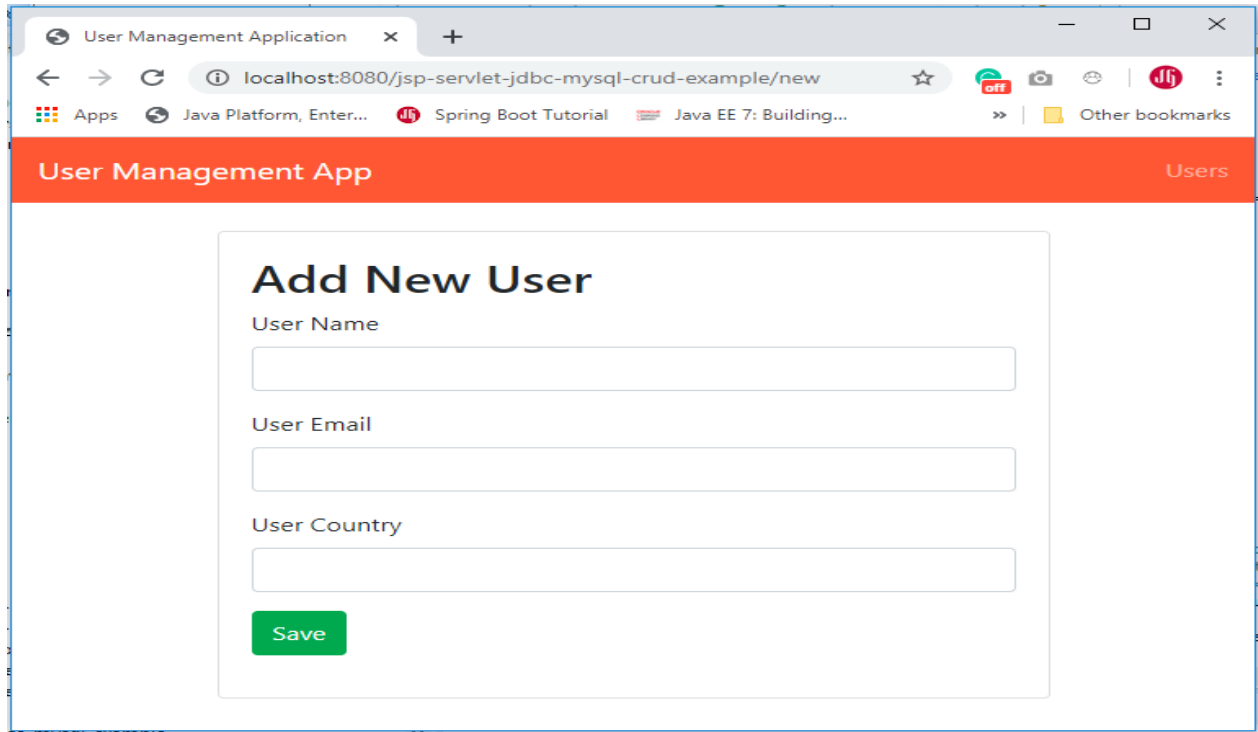
            <fieldset class="form-group">
                <label>User Country</label> <input type="text"
value="<c:out value='${user.country}' />" class="form-control" name="country">
            </fieldset>

            <button type="submit" class="btn btn-success">Save</button>
        </form>
    </div>
</div>
</div>
</body>

```

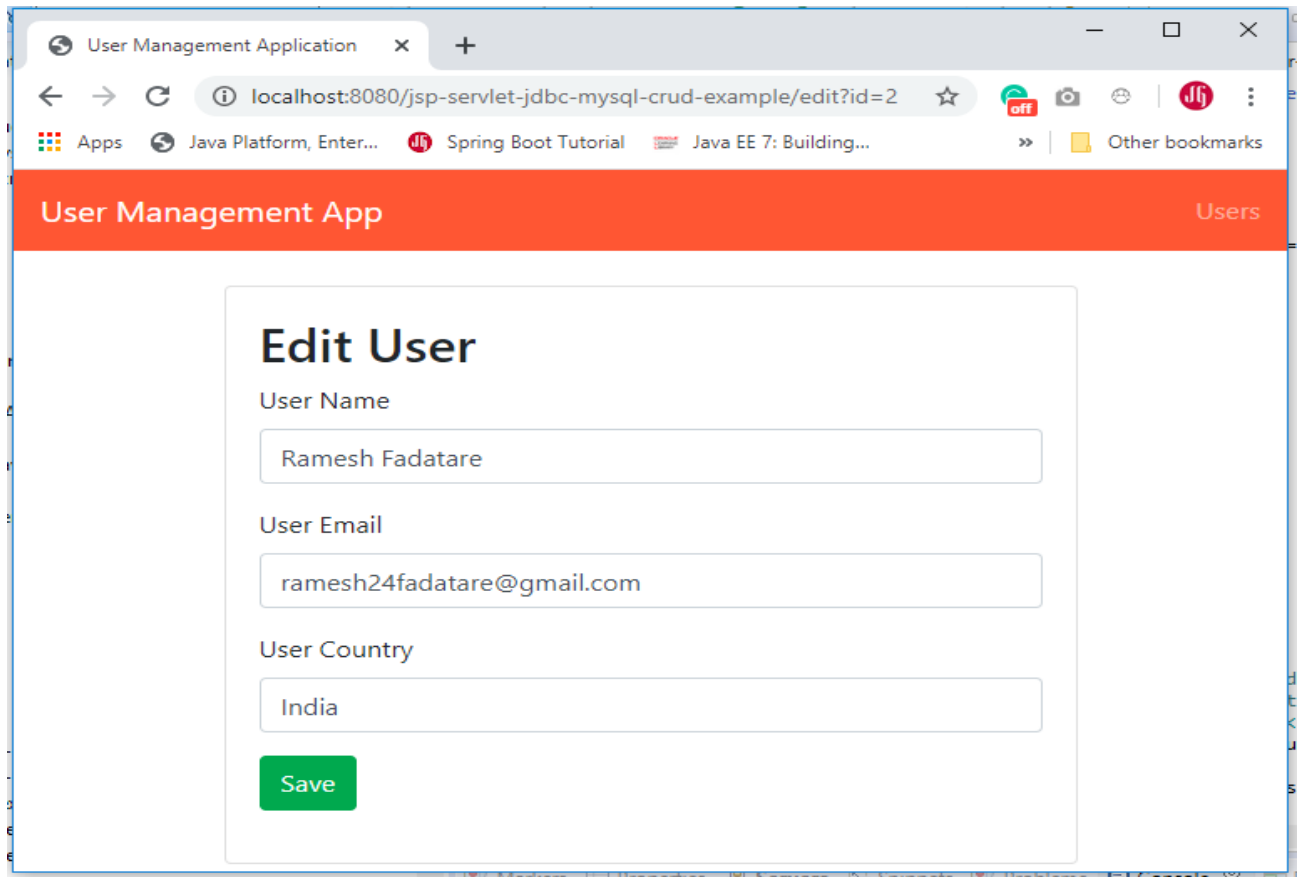
```
</html>
```

Once you will deploy above JSP page in tomcat and open in the browser looks something like this:



The screenshot shows a web browser window with the title 'User Management Application'. The address bar displays 'localhost:8080/jsp-servlet-jdbc-mysql-crud-example/new'. The browser's bookmark bar includes 'Apps', 'Java Platform, Enter...', 'Spring Boot Tutorial', and 'Java EE 7: Building...'. The page features an orange header with 'User Management App' on the left and 'Users' on the right. The main content area contains a form titled 'Add New User' with three input fields: 'User Name', 'User Email', and 'User Country'. A green 'Save' button is positioned below the 'User Country' field.

The above page acts for both functionalities to **create a new User and Edit the same user**. The edit page looks like:



10. Creating Error JSP page

Here's the code of the *Error.jsp* page which simply shows the exception message:

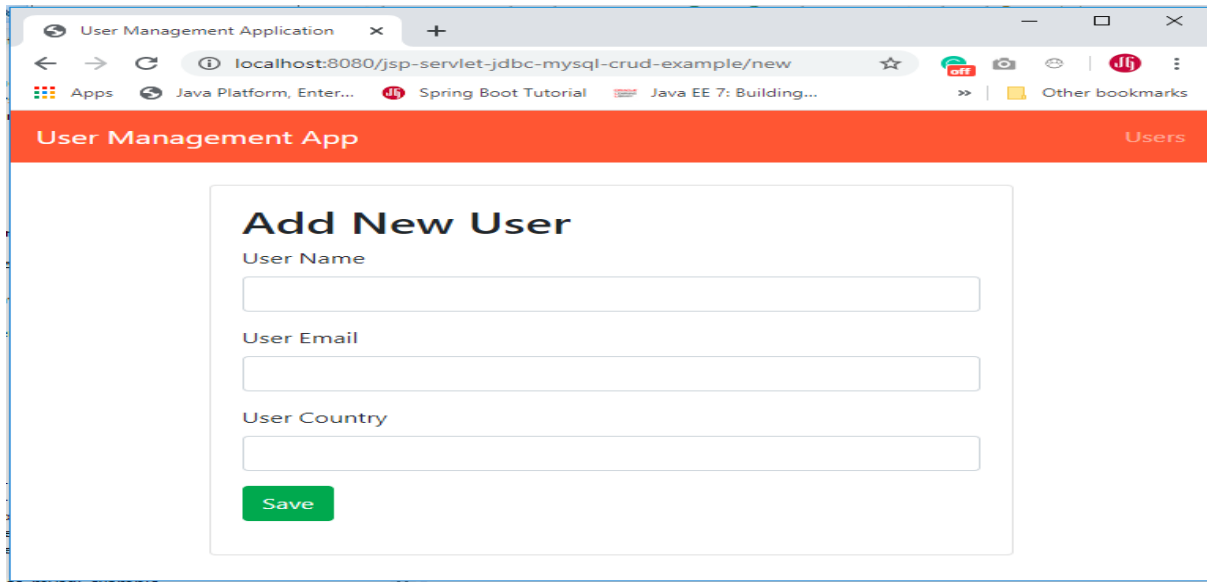
```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" isErrorPage="true" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Error</title>
</head>
<body>
<center>
<h1>Error</h1>
<h2><%=exception.getMessage() %><br/> </h2>
</center>
</body>
</html>
```

11. Deploying and Testing the Application

It's time to see a demo of the above **User Management** web application. Deploy this web application in tomcat server.

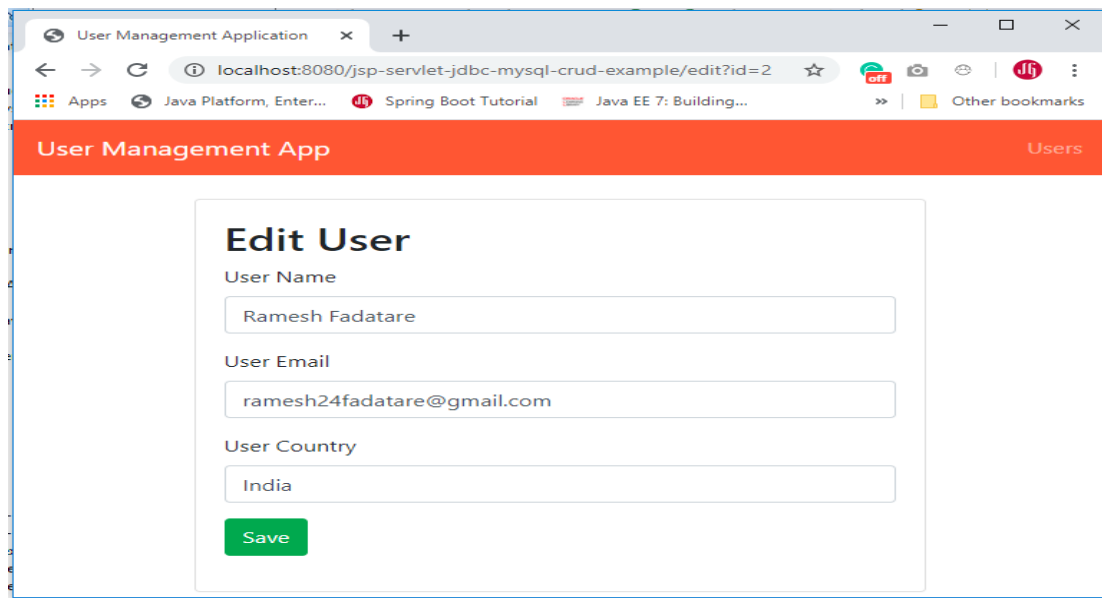
Type the following URL in your web browser to access the **User Management** application: <http://localhost:8080/jsp-servlet-jdbc-mysql-crud-example/>

Create a new User



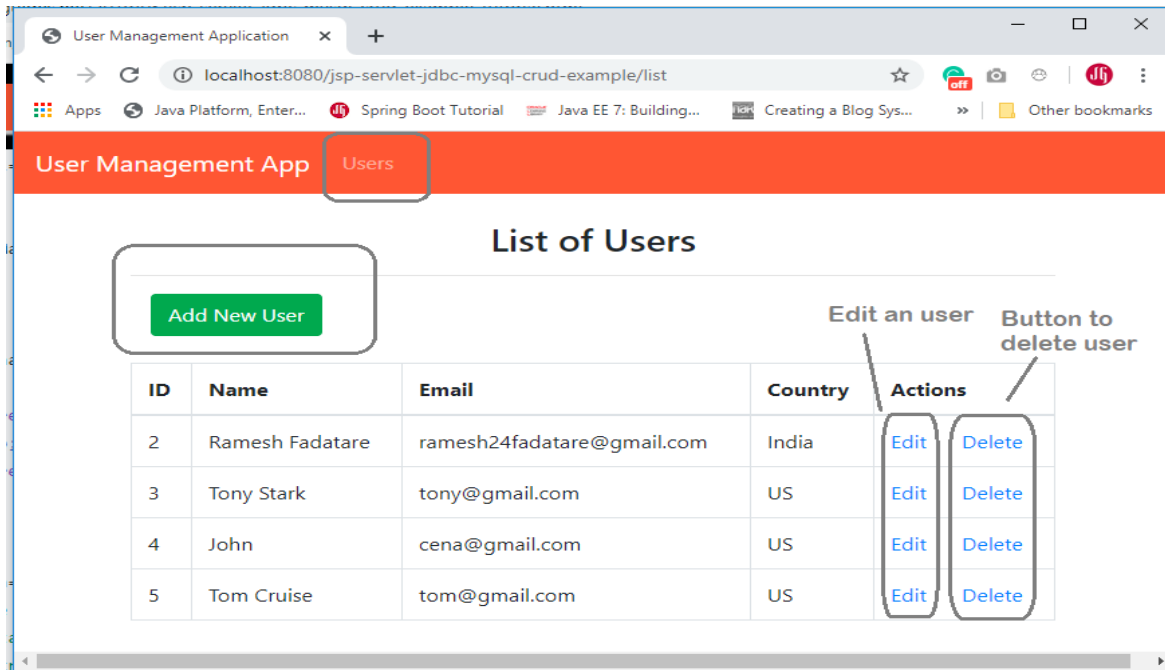
The screenshot shows a web browser window with the title 'User Management Application'. The address bar shows the URL 'localhost:8080/jsp-servlet-jdbc-mysql-crud-example/new'. The page has an orange header with 'User Management App' on the left and 'Users' on the right. The main content area contains a form titled 'Add New User'. The form has three input fields: 'User Name', 'User Email', and 'User Country'. Below these fields is a green 'Save' button.

Edit a User



The screenshot shows a web browser window with the title 'User Management Application'. The address bar shows the URL 'localhost:8080/jsp-servlet-jdbc-mysql-crud-example/edit?id=2'. The page has an orange header with 'User Management App' on the left and 'Users' on the right. The main content area contains a form titled 'Edit User'. The form has three input fields: 'User Name' (containing 'Ramesh Fadatare'), 'User Email' (containing 'ramesh24fadatare@gmail.com'), and 'User Country' (containing 'India'). Below these fields is a green 'Save' button.

List of all Users



GitHub Repository

The source code this tutorial (User Management) is available on my GitHub repository at <https://github.com/RameshMF/jsp-servlet-jdbc-mysql-crud-tutorial>.

Check out [Build Todo App using JSP, Servlet, JDBC, and MySQL](#).

Servlet + JSP + JDBC + MySQL Examples

- [Servlet + JSP + JDBC + MySQL Example](#)
- [Registration Form using JSP + Servlet + JDBC + Mysql Example](#)
- [Login Form using JSP + Servlet + JDBC + MySQL Example](#)

APPENDIX B2: DAO Design Pattern

<https://www.journaldev.com/16813/dao-design-pattern>

DAO stands for Data Access Object. DAO [Design Pattern](#) is used to separate the data persistence logic in a separate layer. This way, the service remains completely in dark about how the low-level operations to access the database is done. This is known as the principle of **Separation of Logic**.

With DAO design pattern, we have following components on which our design depends:

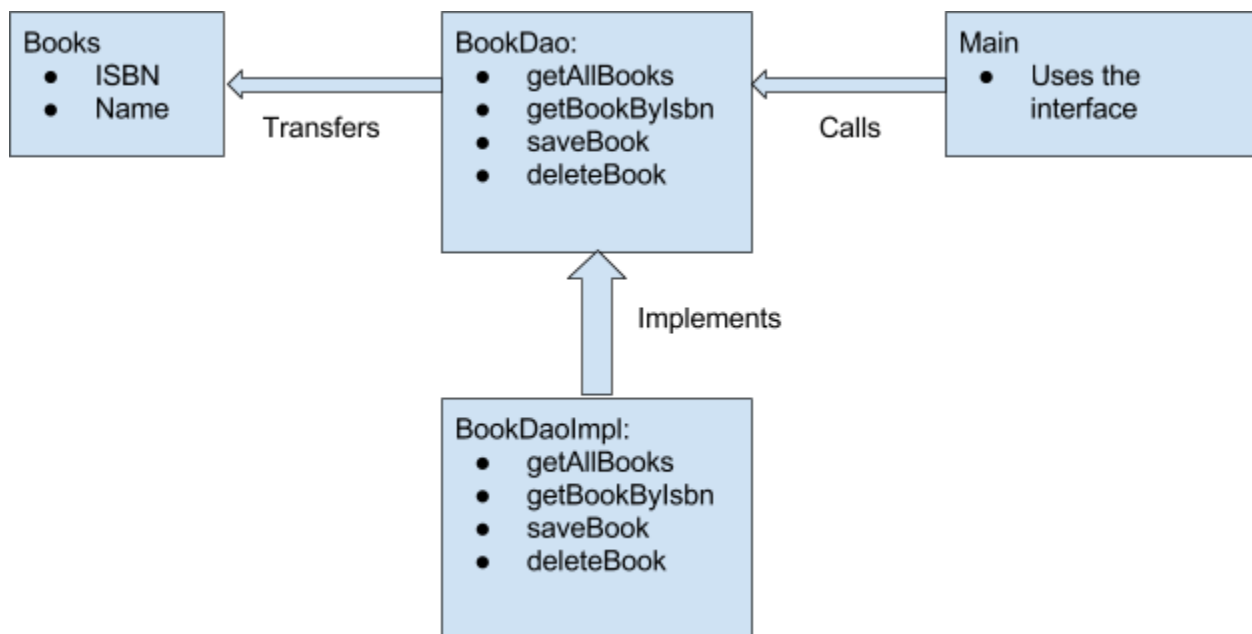
- The model which is transferred from one layer to the other.
- The [interfaces](#) which provides a flexible design.
- The interface implementation which is a concrete implementation of the persistence logic.

Implementing DAO pattern

With above mentioned components, let's try to implement the DAO pattern. We will use 3 components here:

1. The **Book** model which is transferred from one layer to the other.
2. The **BookDao** interface that provides a flexible design and API to implement.
3. **BookDaoImpl** concrete class that is an implementation of the **BookDao** interface.

Let us put this logic into a diagram:



DAO Pattern model Class

Now, let's put up our model object.

```

package com.journaldev.model;

public class Books {

    private int isbn;
    private String bookName;

    public Books () {

```



```

    }

    public Books(int isbn, String bookName) {
        this.isbn = isbn;
        this.bookName = bookName;
    }

    // getter setter methods
}

```

It is a simple object with just 2 properties to keep things simple.

DAO Pattern Interface

Let's define the interface to access the data associated with it at persistence level.

```

package com.journaldev.dao;

import com.journaldev.model.Books;

import java.util.List;

public interface BookDao {

    List<Books> getAllBooks();
    Books getBookByIsbn(int isbn);
    void saveBook(Books book);
    void deleteBook(Books book);
}

```

DAO Pattern Implementation

Next, we create a concrete class implementing the above interface.

```

package com.journaldev.daoimpl;

import com.journaldev.dao.BookDao;
import com.journaldev.model.Books;

import java.util.ArrayList;
import java.util.List;

```

```

public class BookDaoImpl implements BookDao {

    //list is working as a database
    private List<Books> books;

    public BookDaoImpl() {
        books = new ArrayList<>();
        books.add(new Books(1, "Java"));
        books.add(new Books(2, "Python"));
        books.add(new Books(3, "Android"));
    }

    @Override
    public List<Books> getAllBooks() {
        return books;
    }

    @Override
    public Books getBookByIsbn(int isbn) {
        return books.get(isbn);
    }

    @Override
    public void saveBook(Books book) {
        books.add(book);
    }

    @Override
    public void deleteBook(Books book) {
        books.remove(book);
    }
}

```

Using DAO Pattern

Finally, we put this implementation to use in our main() method:

```

package com.journaldev;

import com.journaldev.dao.BookDao;
import com.journaldev.daoimpl.BookDaoImpl;
import com.journaldev.model.Books;

public class AccessBook {

    public static void main(String[] args) {

```

```

        BookDao bookDao = new BookDaoImpl();

        for (Books book : bookDao.getAllBooks()) {
            System.out.println("Book ISBN : " +
book.getIsbn());
        }

        //update student
        Books book = bookDao.getAllBooks().get(1);
        book.setBookName("Algorithms");
        bookDao.saveBook(book);
    }
}

```

Advantages of DAO pattern

There are many advantages for using DAO pattern. Let's state some of them here:

1. While changing a persistence mechanism, service layer doesn't even have to know where the data comes from. For example, if you're thinking of shifting from using MySQL to MongoDB, all changes are needed to be done in the DAO layer only.
2. DAO pattern emphasis on the low coupling between different components of an application. So, the View layer have no dependency on DAO layer and only Service layer depends on it, even that with the interfaces and not from concrete implementation.
3. As the persistence logic is completely separate, it is much easier to write Unit tests for individual components. For example, if you're using JUnit and Mockito for testing frameworks, it will be easy to mock the individual components of your application.
4. As we work with interfaces in DAO pattern, it also emphasizes the style of "work with interfaces instead of implementation" which is an excellent OOPs style of programming.

EXAMPLE 2

```

public class Student {
    private String name;
    private int rollNo;

    Student(String name, int rollNo){
        this.name = name;
        this.rollNo = rollNo;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {

```

```

        this.name = name;
    }

    public int getRollNo() {
        return rollNo;
    }

    public void setRollNo(int rollNo) {
        this.rollNo = rollNo;
    }
}

```

Step 2

```

import java.util.List;

public interface StudentDao {
    public List<Student> getAllStudents();
    public Student getStudent(int rollNo);
    public void updateStudent(Student student);
    public void deleteStudent(Student student);
}

import java.util.ArrayList;
import java.util.List;

public class StudentDaoImpl implements StudentDao {

    //list is working as a database
    List<Student> students;

    public StudentDaoImpl(){
        students = new ArrayList<Student>();
        Student student1 = new Student("Robert",0);
        Student student2 = new Student("John",1);
        students.add(student1);
        students.add(student2);
    }

    @Override
    public void deleteStudent(Student student) {
        students.remove(student.getRollNo());
        System.out.println("Student: Roll No " + student.getRollNo()
+ ", deleted from database");
    }

    //retrive list of students from the database
    @Override
    public List<Student> getAllStudents() {
        return students;
    }

    @Override

```

```

    public Student getStudent(int rollNo) {
        return students.get(rollNo);
    }

    @Override
    public void updateStudent(Student student) {
        students.get(student.getRollNo()).setName(student.getName());
        System.out.println("Student: Roll No " + student.getRollNo()
+ ", updated in the database");
    }
}

public class DaoPatternDemo {
    public static void main(String[] args) {
        StudentDao studentDao = new StudentDaoImpl();

        //print all students
        for (Student student : studentDao.getAllStudents()) {
            System.out.println("Student: [RollNo : " +
student.getRollNo() + ", Name : " + student.getName() + " ]");
        }

        //update student
        Student student =studentDao.getAllStudents().get(0);
        student.setName("Michael");
        studentDao.updateStudent(student);

        //get the student
        studentDao.getStudent(0);
        System.out.println("Student: [RollNo : " +
student.getRollNo() + ", Name : " + student.getName() + " ]");
    }
}

```