Lecture 7

The Model-View-Controller (MVC) Software Architecture

Objectives

The objective of this lecture is to introduce the model view controller software architecture, design and web based application development. After this lecture, student will

- learn how to design model view controller architectures and develop software applications in web-based environment.
- learn the development of web based applications involving two components; front end development and back end development.
- learn how to develop MVC software architecture in client-server systems of web application and connecting database in multi-users system.
- learn the web application development using Java Server Page (JSP) for front end development and Java Servlet for back end server process development and connecting MySQL database for data storage.

Reference Reading

[1] Brahma Dathan and Sarnath Ramnath, "Object-Oriented Analysis, Design and Implementation", An Integrated Approach, Second Edition, 2015, UTiCS, Springer.
  - Chapter 11: Interactive Systems and the MVC Architecture
  - Chapter 12: Designing with Distributed Objects

[2] Ian Sommerville, "Software Engineering", 10th Edition, Pearson Education Limited 2016
  - Chapter 6: Architectural Design
  - Chapter 15: Software Reuse

[3] Roger S. Pressman, Bruce R. Maxim, "Software Engineering: A Practitioner's approach", 9th Edition, © 2020 by McGraw-Hill Education
  - Chapter 10: Architectural Design—A Recommended Approach
  - Chapter 11: Component Level Design

[4] https://krazytech.com/programs/a-login-application-in-java-using-model-view-controllermvc-design-pattern

[5] An MVC Example with Servlets and JSP, https://www.baeldung.com/mvc-servlet-jsp

[6] JSP Servlet MVC Example with Database, https://www.javaguides.net/2020/01/jsp-servlet-mvc-example-with-database.html

1. Introduction

We have seen examples and case-studies involving relatively simple software systems, Library Automation System in previous lecture. This simplicity enabled us to use a fairly general step-by-step approach, viz., specify the requirements, model the behavior, find the classes, assign responsibilities, capture class interactions, and so on.

In larger systems, such an approach may not lead to an efficient design and it would be wise to rely on the experience of software designers who have worked on the problem and devised strategies to tackle the problem. For the problem of creating large software systems, we could reuse a structure or architecture pattern provided by choosing **a software architecture** that we discussed in previous lecture.

In this lecture, we use a well-known software architecture pattern called **the Model–View–Controller or MVC pattern**. Next we design a small interactive system using such architecture, look at some problems that arise in this context and explore solutions for these problems using design patterns. Finally, we discuss pattern-based solutions in software development and some other frequently employed architectural patterns.

2. The MVC Architectural Pattern

The MVC pattern divides the application into **three subsystems**: ***model***, ***view***, and ***controller***. The pattern separates the **application object or the data**, which is termed the **Model**, from the manner in which it is rendered to the **end-user (View)** and from the way in which the end-user manipulates it (**Controller**).This separation of concerns makes the application ***easier to maintain*** and ***extend***, as ***changes to one component do not require changes to the other components***.

2.1 The MVC Implementation

As with the MVC software architecture, the designer needs to have **a clear idea about how the responsibilities are to be shared between the subsystems**. This task can be simplified if the role of each subsystem is clearly defined.

(1) The **view** is responsible for all the **presentation issues**.
(2) The **model** holds the **application object**.
(3) The **controller** takes care of the **response strategy**.

These three functionalities are lumped together in a system resulting in **a low degree of cohesion**. Thus, the MVC pattern helps produce **highly cohesive modules** with **a low degree of coupling**. This facilitates greater **flexibility and reuse**. **MVC** also provides a powerful way to **organize systems that support multiple presentations of the same information**.

2.2 Interaction between MVC Components

The three components of the MVC; the model, view and controller components; shown in the Figure 7.1, communicate flow ensures that each component is responsible for a specific aspect of the application's functionality, leading to a more maintainable and scalable architecture.

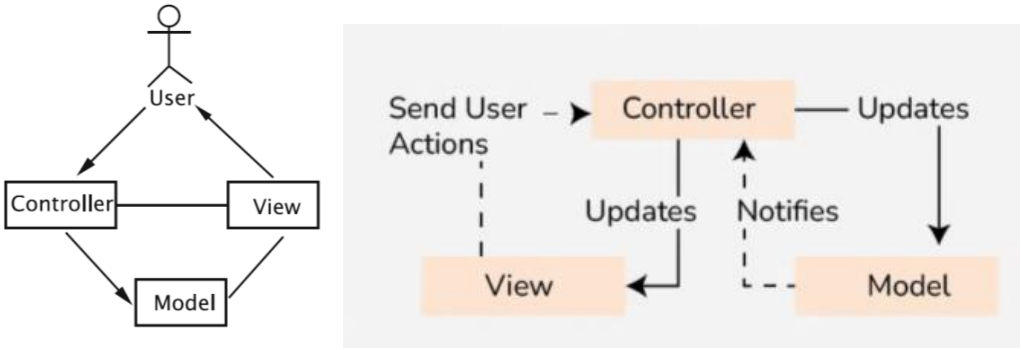


Figure 7.1: The interaction between three components in MVC architecture

In a typical application, the interaction between these three components as follows:

(1) User Interaction with View:
- The user interacts with the View, such as clicking a button or entering text into a form.

(2) View Receives User Input:
- The View receives the user input and forwards it to the Controller.

(3) Controller Processes User Input:
- The Controller receives the user input from the View.
- It interprets the input, performs any necessary operations (such as updating the Model), and decides how to respond.

(4) Controller Updates Model:
- The Controller updates the Model based on the user input or application logic.

(5) Model Notifies View of Changes:
- If the Model changes, it notifies the View.

(6) View Requests Data from Model:
- The View requests data from the Model to update its display.

(7) Controller Updates View:
- The Controller updates the View based on the changes in the Model or in response to user input.

(8) View Renders Updated UI:
- The View renders the updated UI based on the changes made by the Controller.

It is important to **distinguish the UI from the rest of the system**: beginners often **mistake the UI for the view**. In MVC architecture of the system lies behind the UI; **both** the **view** and the **controller are subsystems** at the same level of abstraction that employ components of the UI to accomplish their tasks. The **view subsystem** is therefore **responsible for all the look and feel issues**, whether they arise from a human–computer interaction perspective (e.g., kinds of buttons being used) or from issues relating to how we render the model. Figure 7.2 shows how we might present the MVC architecture while accounting for these practical considerations.

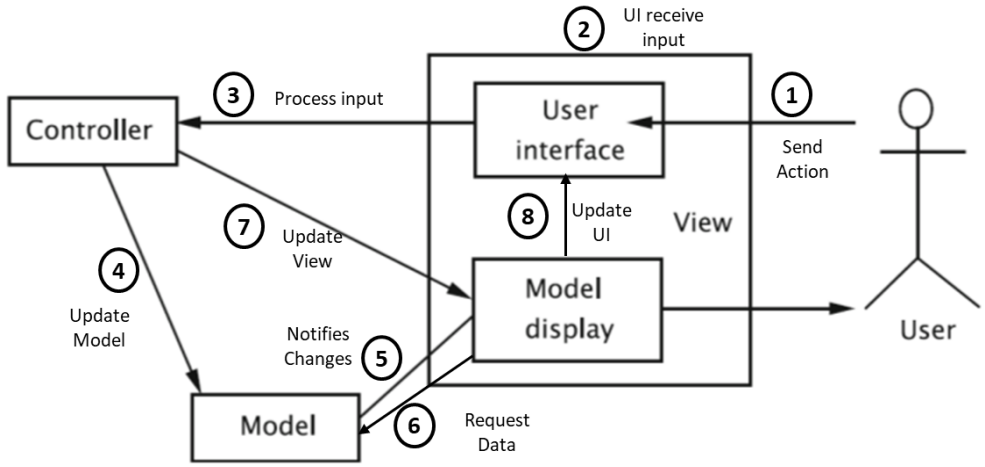


Figure 7.2: An alternate view of the MVC architecture

User-generated events may **cause a controller to change the model, or view, or both**. For example, suppose that the model stored the text that is being edited by the end-user. When the user deletes or adds text, the controller captures the changes and notifies the model. The **view**, which observes the **model**, then **refreshes its display**, with the result that the end-user sees the changes he/she made to the data. In this case, user-input caused a change to both the model and the view. On the other hand, consider **a user scrolling the data. Since no changes are made to the data itself, the model does not change and need not be notified.** But the view now needs to display previously-hidden data, which makes it necessary for the view to contact the model and retrieve information.

More than **one view–controller pair may be associated with a model**. Whenever **user input causes one of the controllers to notify changes to the model, all associated views are automatically updated**. It could also be the case that the model is changed not via one of the controllers, but through some other mechanism. In this case, the model must notify all associated views of the changes.

The **view–model relationship** is that of a **subject–observer**. The model, as the subject, maintains references to all of the views that are interested in observing it. **Whenever an action that changes the model occurs, the model automatically notifies all of these views**. The **views then refresh their displays**. The guiding principle here is that each view is a faithful rendering of the model.

2.3 Simple MVC Architecture Examples

The MVC pattern implementation example using Java, we are required to create the following three classes.

- **The Model layer** is simple Java class **Employee Class**
- **The View Layer** is another Java class **EmployeeView Class**
- **The Control Layer** is another Java class **EmployeeContoller Class**

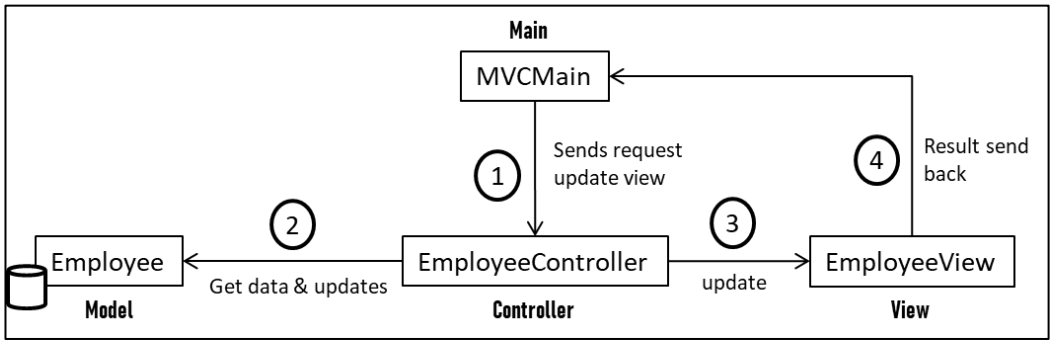


Figure 7.3: MVC example in Java

This separation results in user requests being processed as follows:

(1) Model Layer

The Model in the MVC design pattern acts as **a data layer for the application**. It represents the **business logic for application** and also the **state of application**. The model object fetches and stores the **model state in the database**. Using the model layer, rules are applied to the data that represents the concepts of application.

The first step to implement MVC pattern, the following code **creates a model class** that simply consists of getter and setter methods to the `Employee` class.

Employee.java

```
// class that represents model
public class Employee {

      // declaring the variables
       private String EmployeeName;
       private String EmployeeId;
       private String EmployeeDepartment;

      // defining getter and setter methods
       public String getId() {
          return EmployeeId;
       }
       public void setId(String id) {
          this.EmployeeId = id;
       }
       public String getName() {
          return EmployeeName;
       }
       public void setName(String name) {
          this.EmployeeName = name;
       }
       public String getDepartment() {
              return EmployeeDepartment;
       }
       public void setDepartment(String Department) {
              this.EmployeeDepartment = Department;
       }
    }
```

(2) View Layer

The view represents the **visualization of data received from the model**. The view layer consists of output of application or user interface. It sends the requested data to the client, which is fetched from model layer by controller.

Second, create a view class using the `EmployeeView` class.

EmployeeView.java

```
// class which represents the view
public class EmployeeView {
     // method to display the Employee details
      public void printEmployeeDetails (String EmployeeName,
                  String EmployeeId, String EmployeeDepartment){
       System.out.println("Employee Details: ");
       System.out.println("Name: " + EmployeeName);
       System.out.println("Employee ID: " + EmployeeId);
       System.out.println("Employee Department: " + EmployeeDepartment);
       }
    }
```

(3) Controller Layer

The controller layer gets the **user requests from the view layer and processes them**, with the necessary validations. It acts as **an interface between Model and View**. The requests are then sent to model for data processing. Once they are processed, the data is sent back to the controller and then displayed on the view.

Finally, creates the controller using the `EmployeeController` class.

EmployeeController.java

```
// class which represent the controller
public class EmployeeController {

      // declaring the variables model and view
       private Employee model;
       private EmployeeView view;

      // constructor to initialize
       public EmployeeController(Employee model, EmployeeView view){
          this.model = model;
          this.view = view;
       }

      // getter and setter methods
       public void setEmployeeName(String name){
          model.setName(name);
       }
       public String getEmployeeName(){
          return model.getName();
       }
       public void setEmployeeId(String id){
          model.setId(id);
       }
       public String getEmployeeId(){
          return model.getId();
       }
       public void setEmployeeDepartment(String Department){
              model.setDepartment(Department);
       }
       public String getEmployeeDepartment(){
              return model.getDepartment();
       }

       // method to update view
       public void updateView() {
          view.printEmployeeDetails(model.getName(),
                            model.getId(), model.getDepartment());
       }
    }
```

(4) Main Class Java file

The following example displays the main file to implement the MVC architecture. Here, we are using the MVCMain class.

The MVCMain class fetches the employee data from the method, `retriveEmployeeFromDatabase()` where we have entered the values. Then it pushes those values in the **model**. After that, it initializes the **view** (`EmployeeView.java`). When view is initialized, the **Controller** (`EmployeeController.java`) is invoked and bind it to `Employee class` and `EmployeeView class`. At last the `updateView()` method (method of controller) update the employee details to be printed to the console.

MVCMain.java

```
// main class
public class MVCMain {
  public static void main(String[] args) {
   //fetching the employee record based on the employee_id
   //from the database
      Employee model = retriveEmployeeFromDatabase();

   // creating a view to write Employee details on console
    EmployeeView view = new EmployeeView();
    EmployeeController controller = new EmployeeController(model, view);
    controller.updateView();

    //updating the model data
    controller.setEmployeeName("Nirnay");
    System.out.println("\n Employee Details after updating: ");
    controller.updateView();
  }

  private static Employee retriveEmployeeFromDatabase(){
    Employee Employee = new Employee();
    Employee.setName("Anu");
    Employee.setId("11");
    Employee.setDepartment("Salesforce");
    return Employee;
  }
}
```

Output:

```
Employee Details:
Name: Anu
Employee ID: 11
Employee Department: Salesforce

Employee Details after updating:
Name: Nirnay
Employee ID: 11
Employee Department: Salesforce
```

In this way, we have learned about MVC Architecture, significance of each layer and its implementation in Java.

In our simple example of the Model-View-Controller (MVC) architecture in Java, we've navigated through its core principles, dissecting each component and their interconnected roles. From the **data-centric *Model***, the **interactive *View***, to the **orchestrating *Controller***, MVC stands as a paradigm of efficient software design. We've seen how this architecture brings clarity and structure to Java **web applications, promoting maintainability, modularity, and testability**.

2.4 Benefits of the MVC Pattern

1. **Cohesive modules**:
   Instead of putting unrelated code (display and data) in the same module, we separate the functionality so that each module is cohesive.
2. **Flexibility**:
   The model is unaware of the exact nature of the view or controller it is working with. It is simply an observable. This adds flexibility.

3. **Low coupling**:
   Modularity of the design improves the chances that components can be swapped in and out as the user or programmer desires. This also promotes parallel development, easier debugging, and maintenance.
4. **Adaptable modules**:
   Components can be changed with less interference to the rest of the system.
5. **Distributed systems:**
   Since the modules are separated, it is possible that the three subsystems are geographically separated.

2.5 Disadvantages of the MVC Pattern

- **Complexity**: Implementing the MVC pattern can **add complexity to the code,** especially for simpler applications, leading to **overhead in development.**
- **Learning Curve:** Developers **need to understand the concept of MVC** and **how to implement it effectively,** which may require additional time and resources.
- **Overhead**: The **communication between components** (Model, View, Controller) can **lead to overhead, affecting the performance of the application, especially in resource-constrained environments**.
- **Potential for Over-Engineering**: In some cases, developers may over-engineer the application by adding unnecessary abstractions and layers, leading to bloated and ***hard-to-maintain code***.
- **Increased File Count**: MVC can result in ***a larger number of files*** and classes compared to simpler architectures, which may make the ***project structure more complex and harder to navigate.***

However, the MVC framework is **industry-wide accepted** in the development of **scalable projects**. By studying a MVC example, you would learn how to implement the MVC pattern and create an application with a stellar user interface. So, let's begin with getting acquainted with the MVC design pattern development for our library system.

3. A Web-Based Version of The Library System

Web-based application system is the **client/server architecture systems** where **multiple clients communicate with just one server or multiple servers**.

One major characteristic of a **web-based application system** is that the **client** (the browser), being a general-purpose program, typically does **no application-related computation at all**. **All business logic and data processing take place at the server**. Typically, the **browser receives web pages from the server** in HTML and displays the contents according to the format, a number of tags and values for the tags, specified in it. In this sense, the **browser simply acts as a 'dumb' program** displaying whatever it gets from the application and **transmitting user data from the client site to the server**.

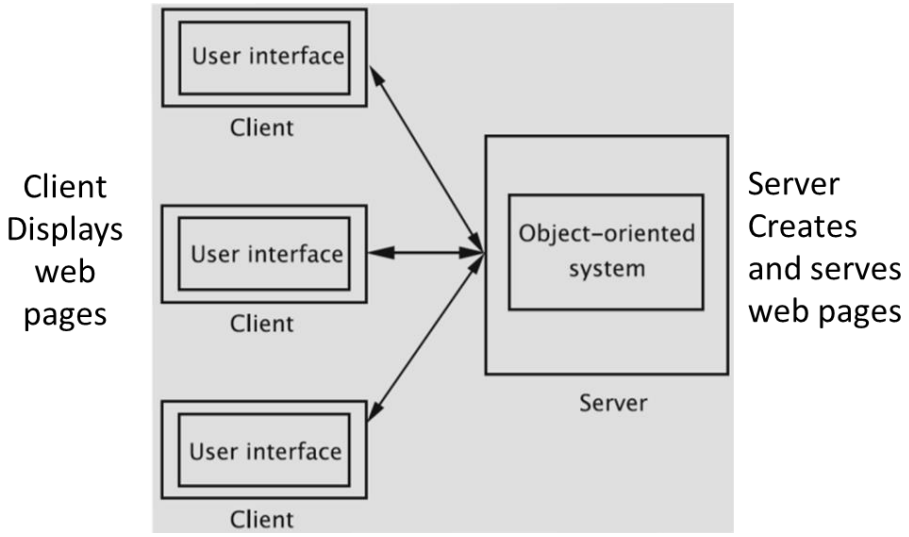


Figure 7.4: Client/Server systems in web based application

The **HTML program shipped from a server to a client** often needs to be customized: the code has to suit the context. For example, when we make **a reservation on a flight**, we expect the system to **display the details of the flight on which we made the reservation**. This requires that HTML code for the screen be dynamically constructed. This is done by code at the server.

In the sense of **layered approach**, for example, **many e-commerce systems** are set up this way:

(1) the **user interface layer** is a web page (perhaps with embedded javascript) viewed by a web browser;

(2) the **business-logic layer** is the software that provides information in response to user requests and processes orders, and

(3) the **database layer** stores information about the products and records user orders.

3.1 Web Application Development

Developing web application system require below skill sets.

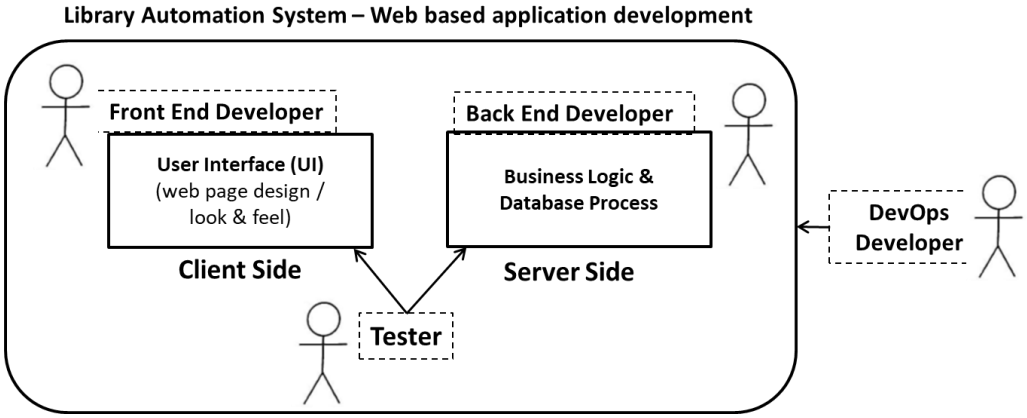


Figure 7.5: Web based application development scope

1. **Front end Developer**
   - **Developing the client side**, the user interface UI of a Library Management System, **a web page** (perhaps HTML page with embedded java script) viewed by a web browser;
2. **Back end Developer**
   - **Developing the server side**, the internal working of the system, such as

- the **business-logic layer** that software provides information in response to user requests and processes orders, and
- the **database layer** stores information about the books and records user borrow, holds information.

3. **Tester**
    - To **test the working of the developed system.**
4. **Devops Developer**
    - To **deploy the Library system and manage the life of the system**.

3.2 Frond end Development

Front-end development primarily focuses on user experience. The front-end developers build the elements of an application that are directly accessed by end-users with a goal of rendering the **entire interface elegant, easy to use, fast, and secure, fostering user engagement and interaction**. The front end app development often focus on specific design elements such as ***text colors and styles, images, graphs and tables, buttons, and overall color schemes***. These elements play a crucial role in **enhancing the visual appeal and user-friendliness of the application.**

Front end app development encompasses various *interactive elements* like **sliders**, **pop-up forms**, and **custom interactive maps**. **An essential part** of a front end application are **navigational menus**, which guide users through the application, enhancing their overall experience and interaction with the website or application. The creation of intuitive and **user-friendly navigational menus** is a key skill for front-end developers.

Client-Side Technologies

Front-end developers require a specific set of skills to **effectively create user interfaces**. This includes proficiency in coding languages like **HTML**, Cascading style sheets (**CSS)**, and **JavaScript (JS),** as well as **a strong understanding of CSS preprocessors** such as Sass and Less and increasingly, jQuery, are essential.

The major trend in front-end development in recent years is the **growth of applications for mobile and smart devices**, with users accessing applications from a growing number of **devices with different screen sizes and interaction options**. As a result, **front-end developers must ensure their application delivers a consistent, high-quality user experience for all devices and usage scenarios.**
Front-End Development Frameworks Front-end frameworks accord you **ready-made code and components, such as prewritten standard functions packaged as libraries**, with no need to build common functionality and components from scratch. **Angular, React, jQuery, Vue.Js, Bootstrap, Semantic UI, Svelte, Preact and Ember.js**. These tools often include drag-and-drop elements, enabling developers to build attractive layouts and applications with various built-in features efficiently.

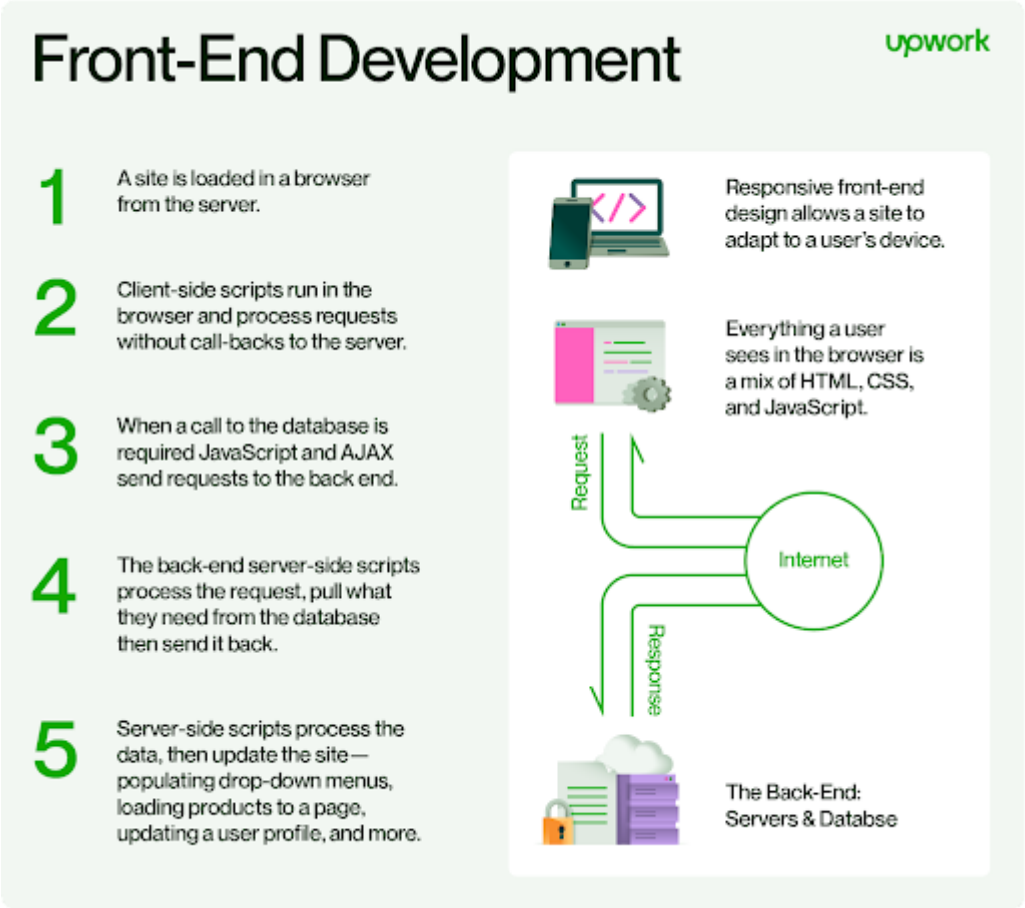


Figure 7.6: Frond end development for web application
(https://www.upwork.com/resources/beginners-guide-back-end-development)

3.3 Back end Development

The back-end is all of the technology required to **process the incoming request and generate and send the response to the client**. This typically includes three major parts:

(1) The **server**. This is the computer that receives requests.
(2) The **app**. This is the application running on the server that listens for requests, retrieves information from the database, and sends a response.
(3) The **database**. Databases are used to organize and persist data.

Backend Development refers to the **server-side development of the web application**. It is the part of the application where the server and database reside and the logics is built to perform operations. It includes the main features and functionalities of the application on the server.

Server-Side Technologies

There are many competing **server-side technologies** which can **generate dynamic contents**: **Java-based** (servlet, JSP, JSF, Struts, Spring, Hibernate), ASP, PHP, Python (Flask, Django), Node.js (JavaScript), (old) CGI Script, and many others.

Java servlet is the **foundation of the Java server-side technology**, **JSP** (Java Server Pages), **JSF** (Java Server Faces), **Struts**, **Spring**, **Hibernate**, and others, are **extensions of the servlet technology**.

For server-side processing, in this lecture we **study Java Server Pages (JSP)** and **Java Servlets.**

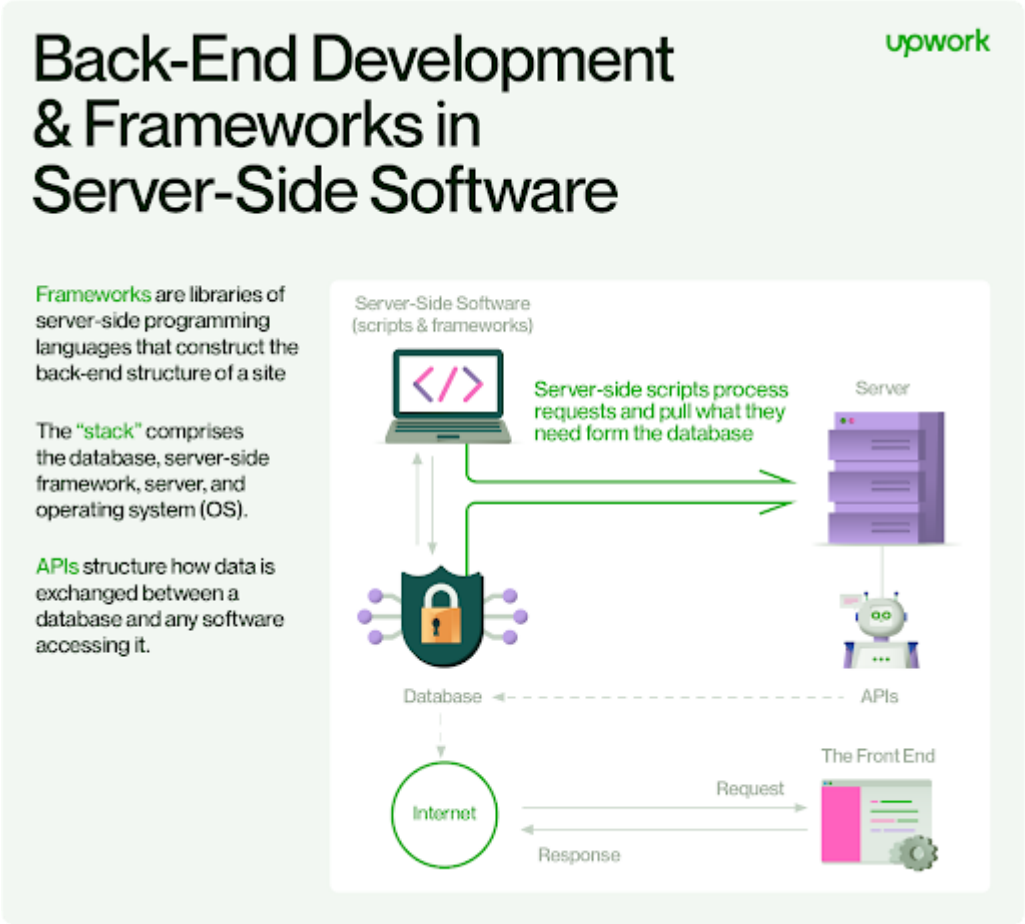


Figure 7.7: Back end development for web application
(https://www.upwork.com/resources/beginners-guide-back-end-development)

4. A Web-Based Version of The Library System

4. 1 Designing User Requirements for Web-based Library System

The first task is to **determine the system requirements for the web application**. We will, as the case study throughout the lecture, restrict the functionality so that the system's size is manageable.

1. The user must be able to type in a URL in the browser and connect to the library system.

2. **Users** are classified into **two categories**: **super-users** and **ordinary members**.

(1) **Super-users** are essentially designated library employees, and
(2) **Ordinary members** are the general public who borrow library books.

The **major difference** between the two groups of users is that

(3) **Super-users** can ***execute any command*** when logged in from a terminal in the library, whereas
(4) **Ordinary members** ***cannot access some 'privileged commands'***.

In particular, the division is as follows:

(a) Only **super-users can issue** the following **commands**: add a member, add a book, return a book, remove a book, process holds, save data to disk, and retrieve data from disk.

(b) **Ordinary members and super-users** may invoke the following commands: **issue** and **renew** books, **place and remove holds**, and **print transactions**.
(c) **Every user** eventually **issues the exit command** to terminate his/her session.

3. Some commands can be **issued from the library only**. These include all of the **commands that only the super-user has access to** and the **command to issue books**.

4. A super-user **cannot issue any commands from outside of the library**. They can log in, but the only command choice will be to **exit the system**.

5. Super-users have special ***User IDs*** and corresponding ***password***. For regular members, their library ***Member ID*** will be their user id and **their phone number** will be the ***password***.

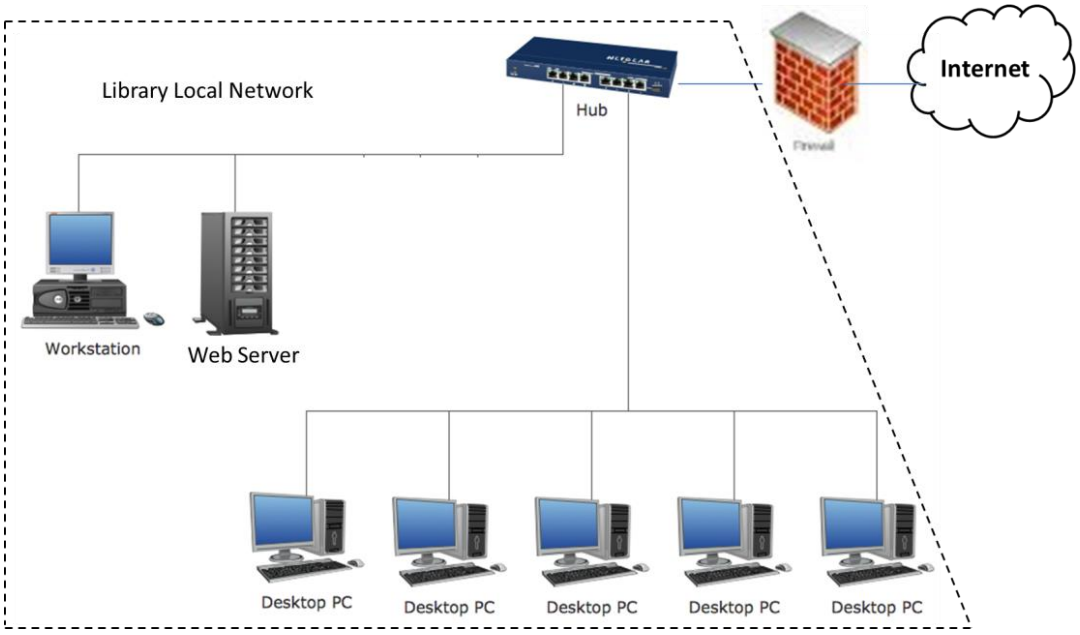


Figure 7.8: Library Local Network

Figure 7.8 shows a library local network system with one server and multiple clients. The web server hosts an object-oriented library system and clients send requests to the server, these requests are processed by the object-oriented system at the server, and the results are returned. The results are then shown to end-users via the user interface at the clients. Super users and members access the system within the library local network.

4.2 Use Case Diagram of Library Management System:

Use case diagram is a **Behaviour model** that describes and displays the **relation or interaction between the users or customers and providers of application service or the system**. It describes different actions that a system performs in collaboration to achieve something with one or more users of the system. Use case diagram for the web based system is described in the Figure 7.9, the difference between previous version is users of the system. According to the requirements, there are two types of users; each can perform actions with their ***privileged functions.*** The **internal process is same as previous system**. However, **in web application** the data store and retrieve is done **using database** whereas **previous project** uses **object storage** and load **to and from the file**.

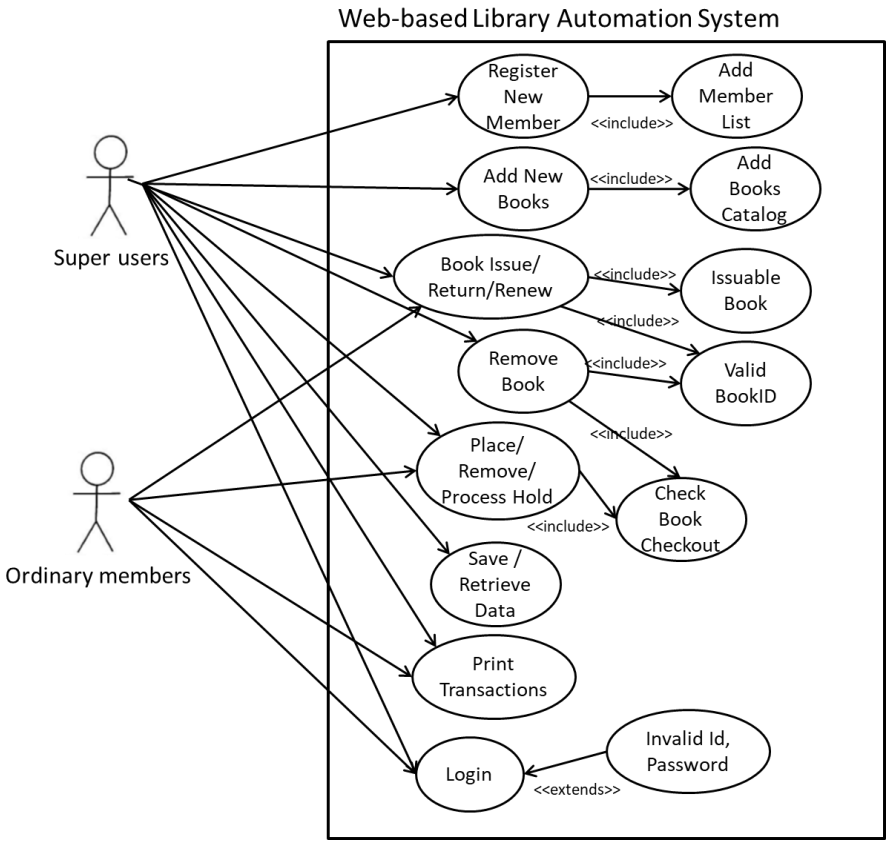


Figure 7.9: Use case diagram for the web-based Library Automation System

4.3 Interface Requirements

An arbitrarily **large number of sequences of interactions are possible between the user and the interface** for the graphical user interface of web application.

Suppose that users be able to abandon most operations in the middle; for example, the user may decide to place a hold on a book, but when the screen to enter the book id and duration of the hold pops up, the user may change her mind and decide not to place a hold. For a second example, **a book may be self-issued** or the member may ask **a library staff member to check out the book**. In the latter case, the member id needs to be input, whereas in the former case, that information is already available to the system.

Therefore, employing the **use-case model** alone to determine the requirements would necessitate the **use of too many conditionals**, and the resulting **sequence diagrams are not easily understood**. So, we describe the requirements mostly through **state transition diagrams**. However, **a single transition diagram** is ***too large and unwieldy***. Therefore, we **split the state transition diagram into a number of smaller ones**. (A little later, we will depict the flow using a sequence diagram as well.)

4.3.1 Logging in and the Initial Menu

In Figure 7.10, we show the **process of logging in to the system**. When the user types in the URL to access the library system, the log in screen that asks for the ***User Id*** and ***Password*** is displayed on the browser. If the user types in a bad user id/password combination, the system presents the log in screen again with an error message. On successful validation, the system displays a menu. What is in the menu depends on whether the **user is an ordinary member or a super-user** and whether the **terminal is in the library or is outside**.

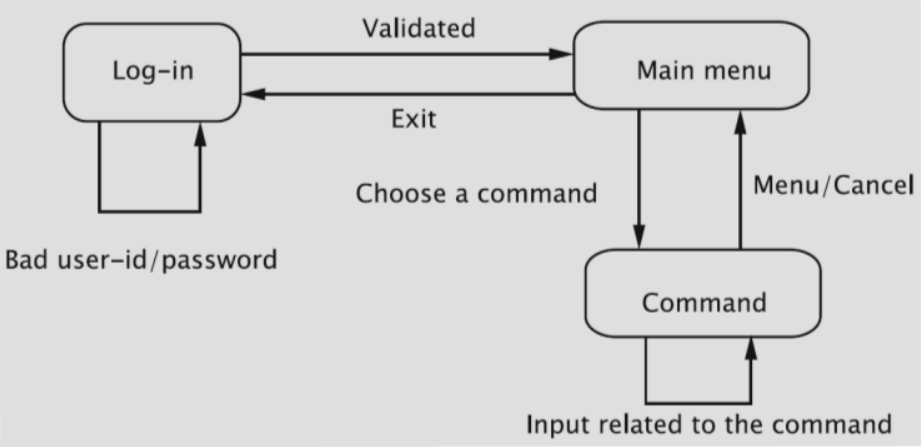


Figure 7.10: State transition diagram for logging in

1. The **Issue Book** command is available only if the user logs in from **a terminal in the library**.
2. **Commands** to **place** a **hold**, **remove** a hold, **print** transactions, and **renew** books are available to members of the library (**not super-users**) **from anywhere**.
3. Certain **commands** are available **only to super-users** who log in **from a library terminal**: these are for **returning or deleting** books, **adding members and books**, **processing holds**, and **saving data** to and **retrieving data from disk**.

A **super-user has to be logged in from a terminal in the library**, or the **menu** will simply contain the **command to exit the system**. The Command State in Figure 7.10 denotes the general flow of a command. When a certain command is chosen, we enter a state that represents the command. How the transitions take place within a command obviously depends on what the command is. All screens allow an option to cancel and go back to the main menu. If this option is chosen, the system goes on to display the main menu awaiting the next command. When the exit command is chosen, the **system logs the user out** and presents the **log in screen again**.

4.3.2 Adding Book

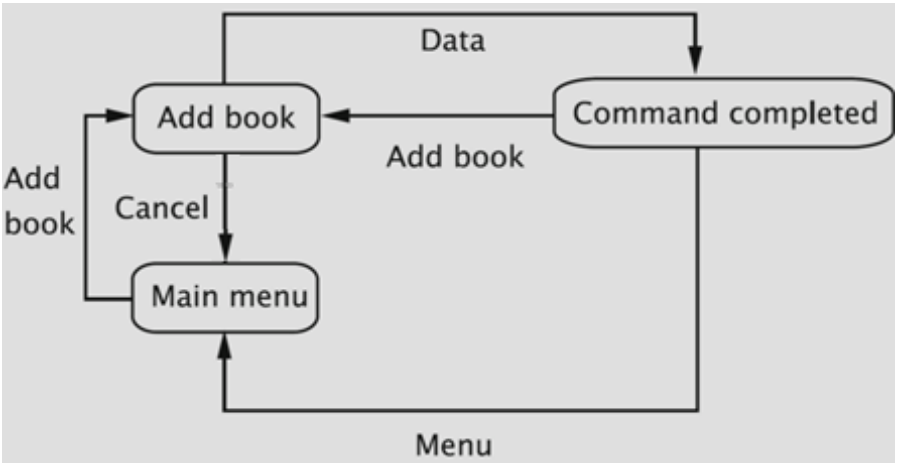


Figure 7.11: State transition diagram for add book

In Figure 7.8, when the **command to add a book is chosen**, the system constructs the initial screen to **add a book**, which should contain **three fields** for entering the ***title***, ***author***, and ***id of the book***, and then display it and enter the Add Book state. By clicking on **a button**, it should be possible for the user ***to submit these values to system***. The system must then call the appropriate method in the **`Library` class** to create a **`Book` object** and enter it into the **`catalog or database`**. The **result of the operation** is **displayed** in the **Command Completed State**. From the Command Completed state, the

system must allow the user to **add another book** (**or) go back to the menu**. In the Add Book state, the user has the **option to cancel the operation** and **go back to the main menu**.

4.3.3 Add Member, Return Book, Remove Book

The requirements are similar to the ones for adding books. We need to accept some input (member details or book id) from the user, access the `Library` object to invoke one of its methods, and display the result. So we do not describe them here nor do we give the corresponding state transition diagrams.

4.3.4 Save Data and Retrieve Data

When the data is to be written to disk, no further input is required from the user. The system should carry out the task and print a message about the outcome. The state transition diagram is given in Figure 7.12. The requirements for "Retrieve Data" are similar to those for saving data.

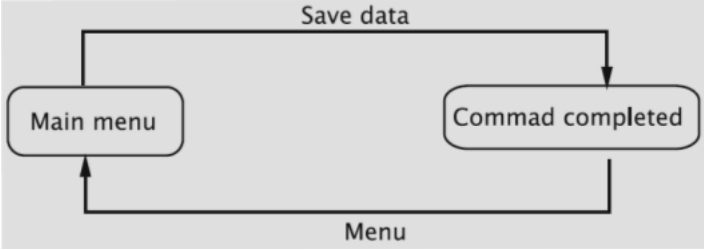


Figure 7.12: State transition diagram for saving data

4.3.5 Issue Book

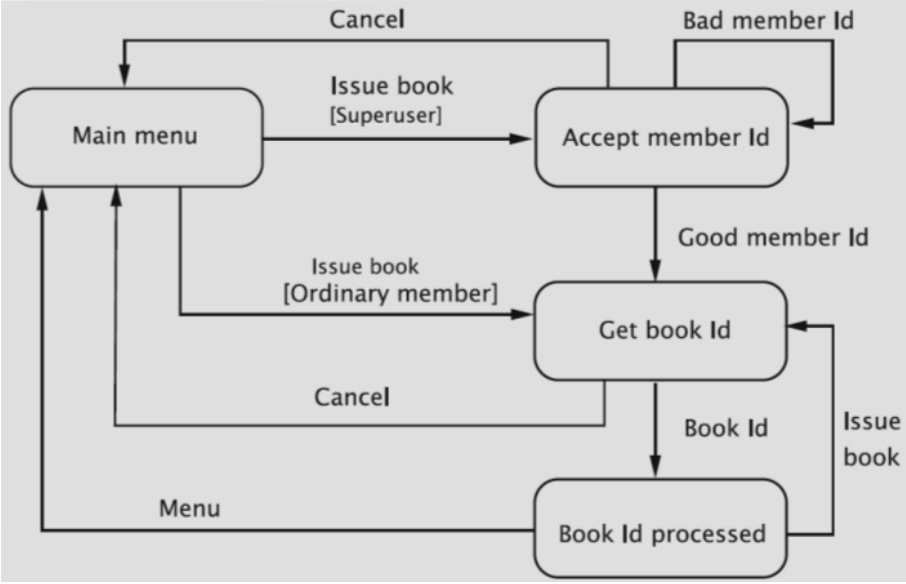


Figure 7.13: State transition diagram for issuing books

In Figure 7.13, a book may be checked out in two different ways:

- **First**, **a member is allowed to check it out himself/herself**.
  In this case, the system already has the user's member id, so that should not be asked again.
- **Second**, he/she may **give the book to a library staff member,** who checks out the book for the member.
  In this case, the library staff member needs to input the member id to the system followed by the book id.

After receiving a book id, the system must attempt to check out the book. Whether the operation is successful or not, the system enters the Book Id Processed state. The complexity arises from the fact that any number of books may be checked out. Thus, after each book is checked out, **the system must ask if more books need to be issued or not**. The system must either go to the Get Book Id state for one more book id or to the Main Menu state. As usual, it should be **possible to cancel the operation at any time**.

4.3.6 Place Hold, Remove Hold, Print Transactions

The requirements for these are similar to those for issuing a book, so we omit their description. Students can do as an assignment for these requirements.

4.3.7 Renew Books

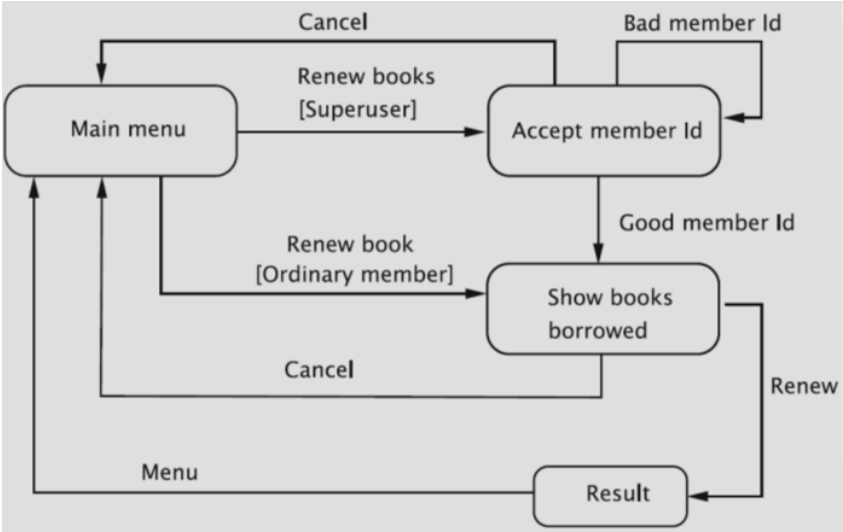


Figure 7.14: State transition diagram for renewing books

For renewing book, as shown in Figure 7.14, the system must **list the title and due date of all the books loaned to the member**. For each book, the system must also present a choice to the user to renew the book. After making the choices, the member clicks a button to send any renew requests to the system. For every book renewal request, the system must display the title, the due date (possibly changed because of renewal), and a message that indicates whether the renewal request was honored. After viewing the results, the member uses a link on the page to navigate to the main menu.

5. Implementation the Web-based Library System

To deploy the system on the web, we need the following:

1. **Classes** associated with the library, which we developed in previous lecture-3; you will recall that this includes classes such as **`Library`**, **`Member`**, **`Book`**, **`Catalog`**, and so on.
2. **Permanent data** (created by the save command) that **stores information** about the members, books, who borrowed what, holds, etc.
3. **HTML files** that support a **GUI for displaying information on a browser** and collecting data entered by the user. For example, when a book is to be returned, a screen that asks for the book id should pop up on the browser. This screen will have a prompt to enter the book id, a space for typing in the same, and a button to submit the data to the system.
4. **A set of files** that **interface between the GUI ((3) above) and the objects** that actually do the processing ((1) above). Servlets will be used to accomplish this task.

5.1 Technology Used for Implementation

Developing web app needs the client-side processing and server-side processing, thus, first, chooses the technology both for the client and server side develop. There are many technologies available for both processing -

- For **client-side processing**, the technologies such as **HTML, CSS, Java Script**
- For **server-side processing**, there are competing technologies such as **Java Server Pages (JSP) and Java Servlets, Active Server Pages (ASP), and PHP.**

To solve the traditional issue we are building a Web development project of library management system in which we will be providing **User-friendly interface for easy navigation**, **seamless book issuance and return policy**, **automated tracking of library activities**, **Regular maintenance of book availability records** and **secure login and access control managed by the admin**.

Here we are developing our project - a Library Management System using **JSP, Java Servlets** and **MySQL** involves a structured methodology MVC (model, view, controller) framework.

1. The **model (DAO)** consists of application data and business rules.
2. The **controller (Servlet)** acts as an interface between views and the model. It mediates input, converting it to commands for the model or view.
3. A **view (JSP)** can be any output representation of data, such as a chart or a diagram, generally HTML or JSP page.

Here **DAO is the Data Access Object** – This part concentrates on business logic and database server connections and operations.

Below are the detailed description about the technology used and methods we are applying in our project.

In Our project all required software's for implementing the Library System:

1. **XAMPP software** (for mysql and Tomcat webserver)
2. **Eclipse IDE** – Integrated development environment for Java applications
3. **HTML/CSS/JavaScript** for client-side programming
4. **Java Servlet and JSP** for server-side Java programming
5. **SQL, MySQL Database System, JDBC** (Java Database Connectivity)

After we downloaded the above required software now we will start creating our project. In the following article, we will discuss about different modules compiled with same category.

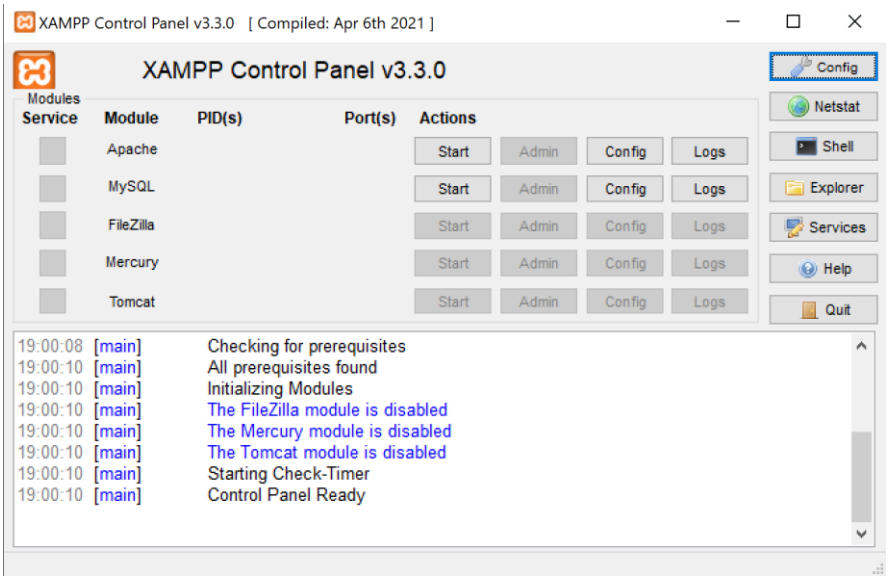
5.1.1 Setting Up

Software requirement for the project development are downloaded and install as follows:

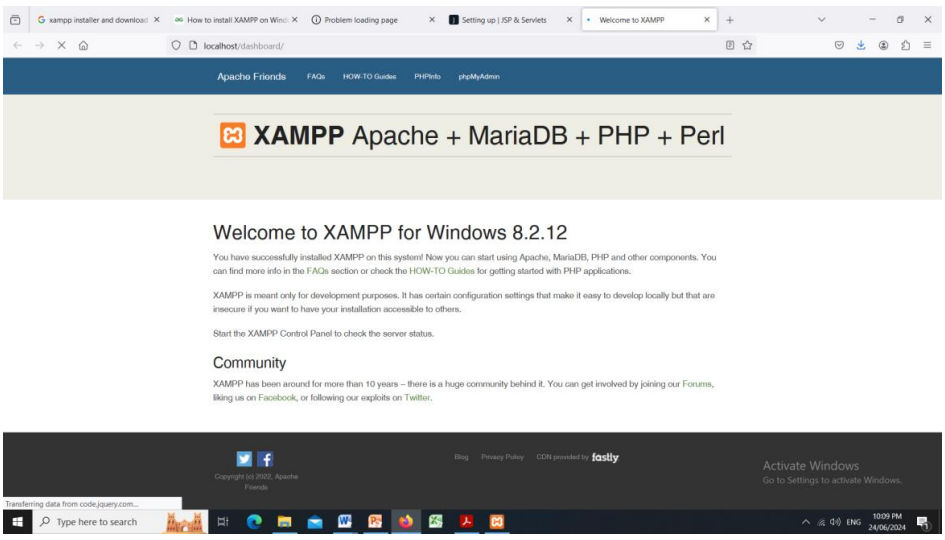
1) Download the XAMPP software from the site https://www.apachefriends.org/downloads.html (for window download - xampp-windows-x64-8.2.12-0-VS16-installer.exe). Then install XAMPP in C: directory.

Please refer to https://www.geeksforgeeks.org/how-to-install-xampp-on-windows/ for detail installation steps.

- After installation XAMPP, open XAMPP control panel as shown below. Then start the Apache server, MySQL server and Tomcat server by clicking "Start" button to test the installation successful.



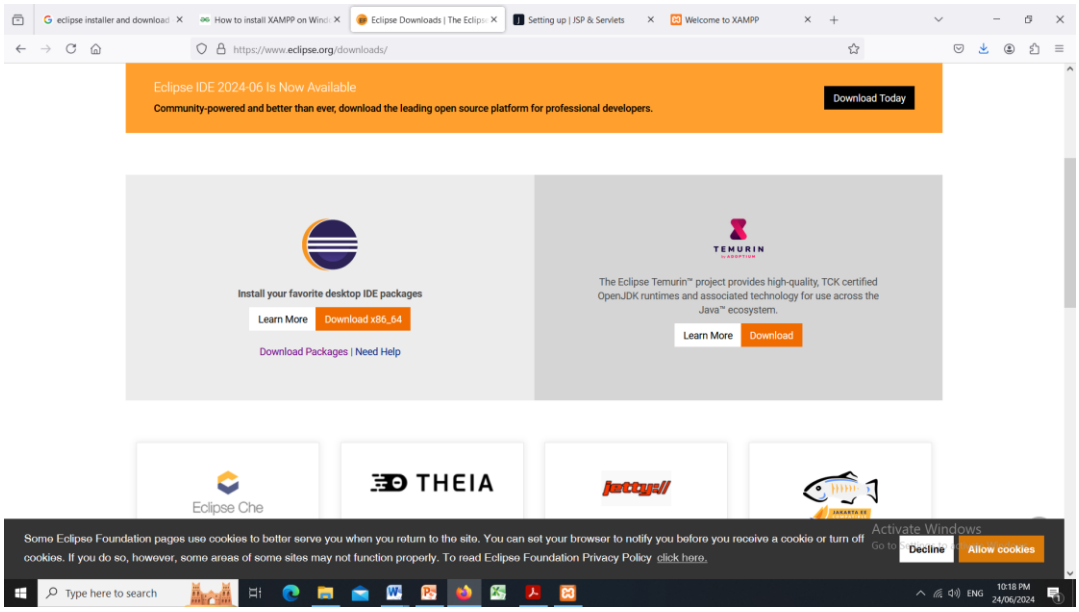
- Then open browser and type "**localhost**" in URL to see the XAMPP is working as shown below.



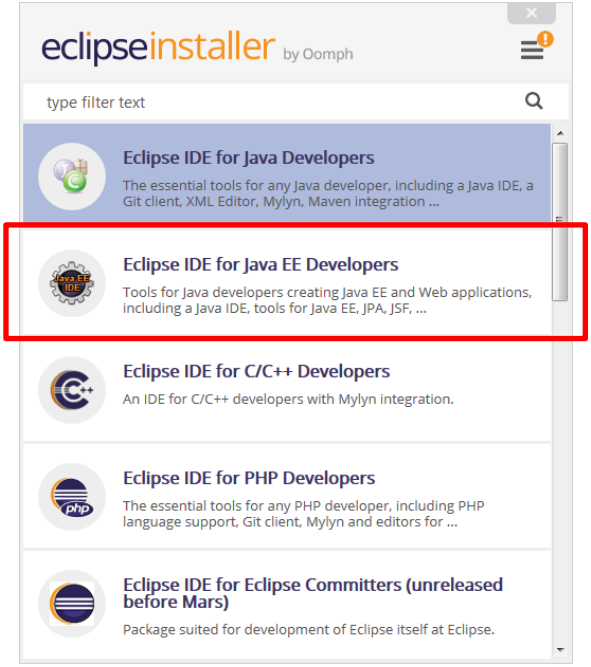
- When the **XAMPP dashboard** is displayed as above, the installation of XAMPP is completed.

2) Download the Eclipse IDE from https://eclipse.org/downloads. (eclipse-inst-jre-win64.exe). Then install by clicking eclipse.exe file.

Please refer to https://www.geeksforgeeks.org/how-to-download-and-install-eclipse-on-windows/ for detail installation steps.



Choose "**Eclipse IDE for Java EE Developers**" for web applications as shown below.

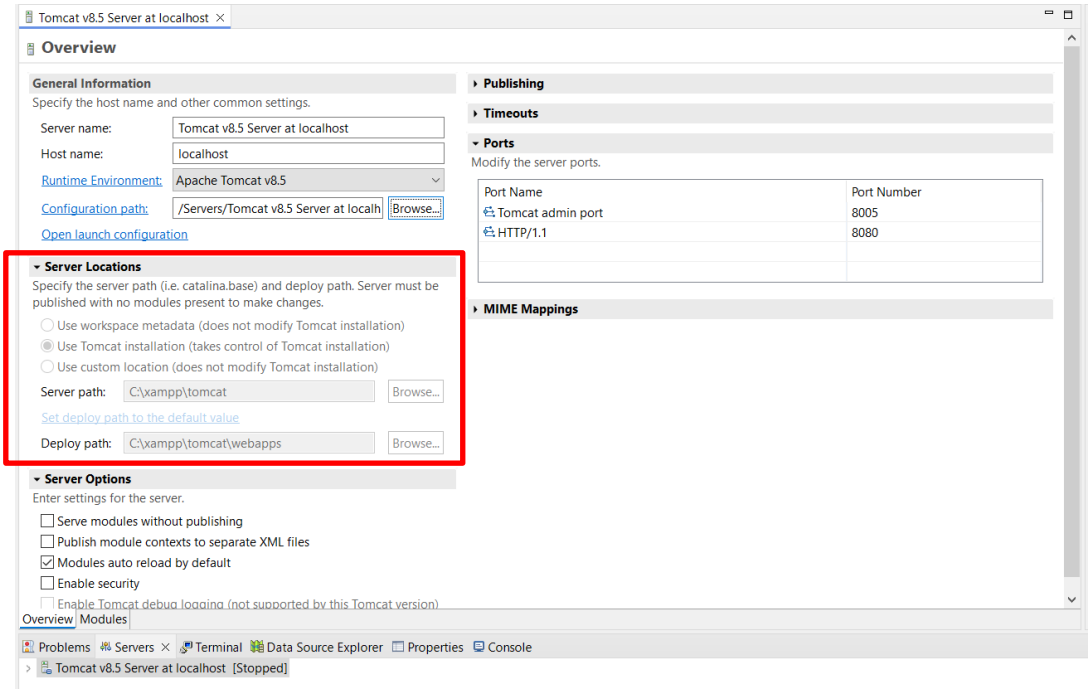


3) Setting up the Eclipse IDE to link with XAMPP tomcat server and deployment paths.

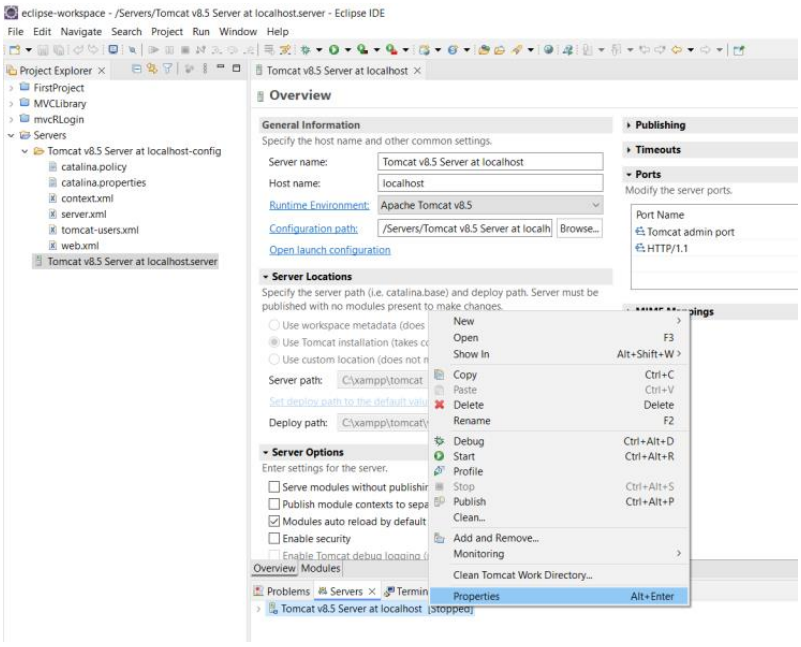
Please refer to https://www.javatpoint.com/how-to-configure-tomcat-server-in-eclipse-ide for detail steps.

- Open the Eclipse
- In **Server tap**, click to create a new server

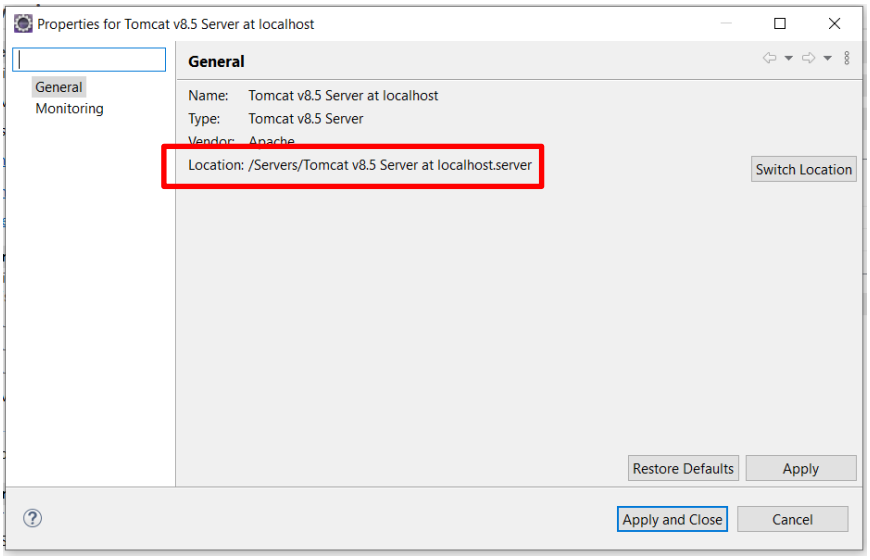
- Choose **Apache**
- Select tomcat version that is installed in your system: **Tomcat 8.5 Server**
- Browse the directory and select **JAVA SE in your system** to **remove default JRE**.
- Select **tomcat folder** from your system.
    - C:\XAMPP\tomcat
- **Click on finish.**
- **Double click the Tomcat v8.5 Server at localhost [Stopped] in Server tap.** The server information shows as below:



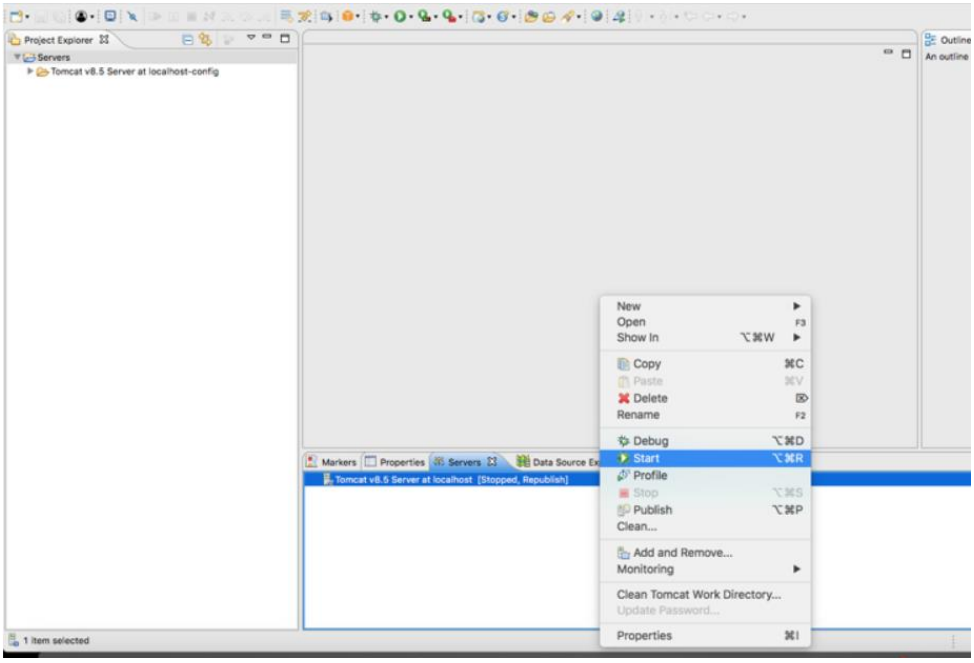
- In server locations, choose “Use Tomcat installation (takes control of Tomcat installation)”
- In Server path: set it to “C:\xampp\tomcat”
- In Deploy path: set it to “C:\xampp\tomcat\webapps”

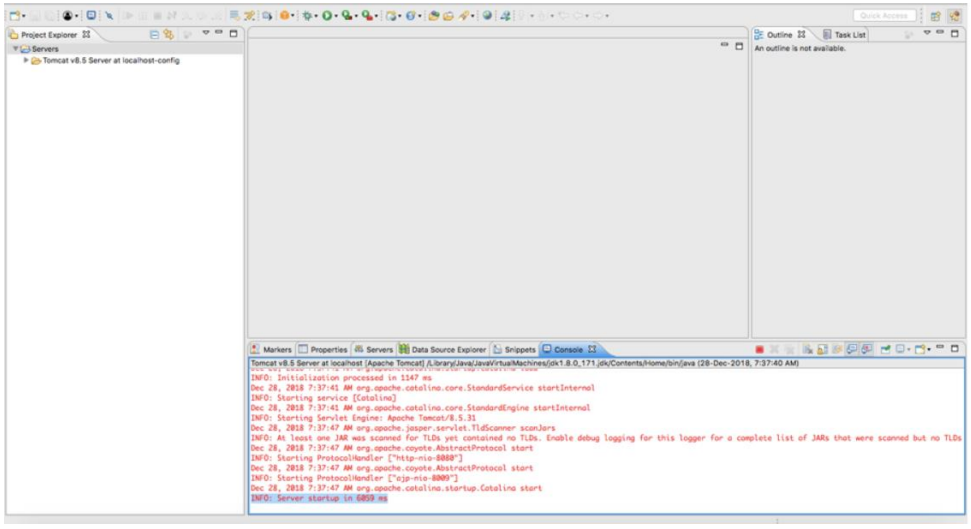


- Then right click on the server and choose properties.
    - Click “Switch location” to “/Servers/Tomcat v8.5 Server at localhost.server”
    - Click “Apply and Close”



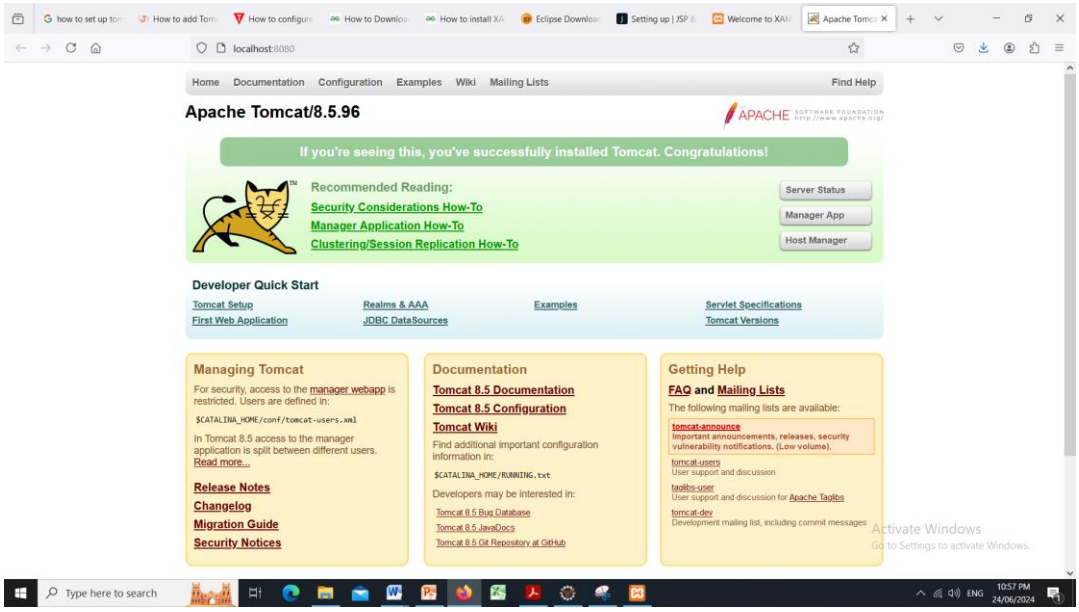
- Finally, test the server by right click on the server and select “Start”





- Server state will change from “Stopped” to “Started”.

- Now check by hitting the url: “localhost:8080/”
- The Apache Tomcat/8.5.96 is showed as below.
  (Here localhost: is the IP address of your local machine and 8080 is the port on which tomcat listen requests.)



The setting up the Eclipse and tomcat server is completed.

5.1.2 Java Server Page (JSP)

In Java, JSP stands for **Jakarta Server Pages**((JSP; formerly **Java Server Pages**)). It is a **server-side technology** which is used for **creating web applications**. It is used to create **dynamic web content**. JSP consists of both **HTML tags and JSP tags**. JSP tags are used to **insert JAVA code into HTML pages**.

JSP is **first converted into a servlet** by the JSP container before processing the client’s request. JSP has various features like JSP Expressions, JSP tags, JSP Expression Language, etc.

Please refer to detail lecture on JSP at https://www.geeksforgeeks.org/introduction-to-jsp/, https://www.tutorialspoint.com/jsp/index.htm, https://www.javatpoint.com/jsp-tutorial. Here we will explain some usage of JSP. In figure 7.15, the lifecycle of a JSP page.

The JSP pages follow these phases:

- Translation of JSP Page
- Compilation of JSP Page
- Classloading (the classloader loads class file) - JSP is converted to a class so this test.jsp is converted to a java class.
- Instantiation (Object of the Generated Servlet is created). Java class is basically a servlet. Every jsp is a servlet.
- Initialization (the container invokes jspInit() method).
- Request processing (the container invokes _jspService() method).
- Destroy (the container invokes jspDestroy() method).

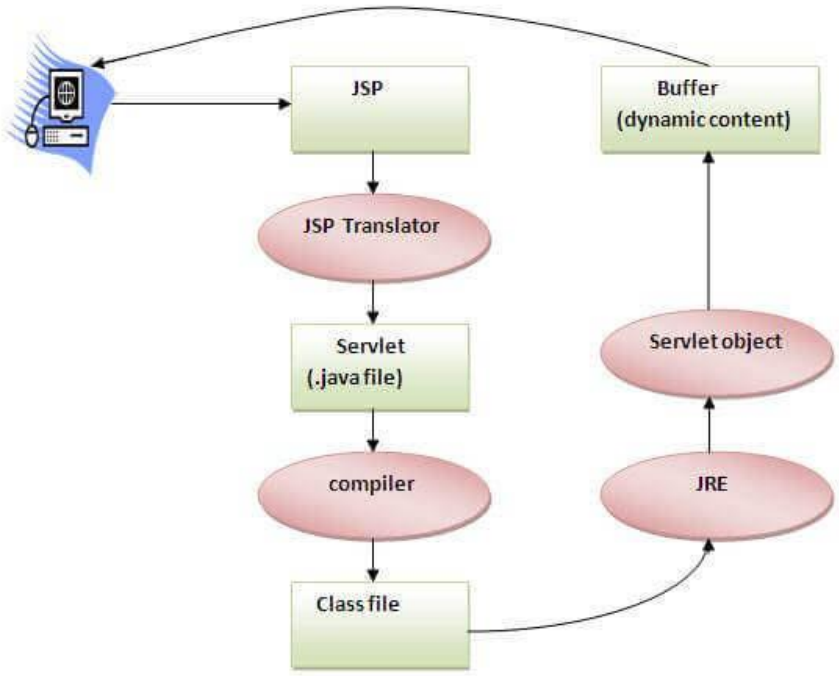


Figure 7.15: The Lifecycle of a JSP Page

When we run it on server (tomcat), tomcat converts jsp to servlet. Default is **get() method.** Every <%...%> is converted added to **doGet() method** of the servlet class is generated from JSP. All the HTML code will be put into out.write() by tomcat. That's why method declared inside <%..%> gives the compilation as we cannot define a method inside a method for that we need : <%!..%>.

(1) Writing Java Code in JSP: Way 1: using <% . .... %>

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;  charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
      <h1>Hello</h1>
      <%
            int i = 3;
            int j = 8;
            int k = 3 + 8;
            out.println("Value of k is " + k);
      %>

</body>
</html>
```

(2) Writing Java Code in JSP: Way 2: using <% = %>

Example 1:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
	<h1>Hello</h1>
	<%
		int i = 3;
		int j = 8;
		int k = i + j;
		out.println("Value of k is " + k);
	%>

	<%
		int l = 3;
		int m = 12;
		int n = l + m;
	%>
	Value of n is
	<%=n%>

</body>
</html>
```

The output of JSP page.

Example 2:

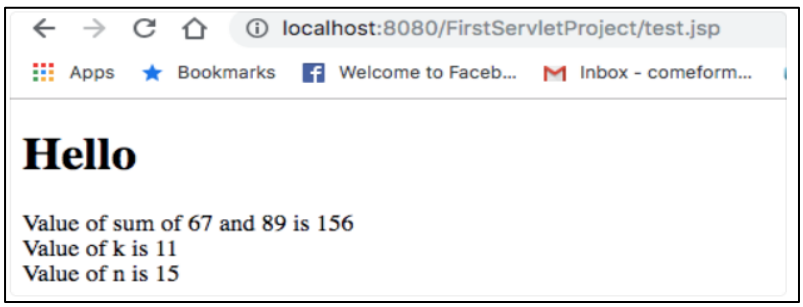
```
<%@ page language="java" contentType="text/html; charset=UTF-8"
	pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
	Simple for loop
	<br>
	<% for(int i = 0; i<5; i++) { %>

		<br>
		Value of i is = <%=i %>

	<%} %>

</body>
</html>
```

The output of JSP page.

(3) Writing Java Code in JSP: Way 3: using <%!...%>

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
        <h1>Hello</h1>
        <%!public int sum(int a, int b) {
                return a + b;
        }%>

        Value of sum of 67 and 89 is
        <%=sum(67, 89)%>
        <br>
        <%
                int i = 3;
                int j = 8;
                int k = i + j;
                out.println("Value of k is " + k);
        %>
        <br>

        <%
                int l = 3;
                int m = 12;
                int n = l + m;
        %>
        Value of n is
        <%=n%>

</body>
</html>
```

The output of JSP page.



(4) Want to use Java Date() to display current date on JSP page:

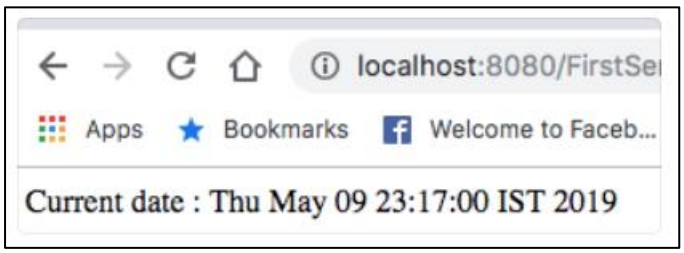
Way 1: Add in the existing page directive:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8" import="java.util.Date" %>
```

Example

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8" import="java.util.Date" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
      Current date :
      <%= new Date() %>

</body>
</html>
```

The output of JSP page.



(5) Want to use Java Date() to display current date on JSP page:

Way 2: Add a separate import statement:

```
<%@ page import="java.util.Date" %>
```

Example

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8"  %>

<%@ page import="java.util.Date" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
      Current date :
      <%= new Date() %>

</body>
</html>
```

(6) To include other JSP pages (header and footer):

```
<%@ include file="/Header.jsp"%>
```

Example

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8"%>

<%@ page import="java.util.Date"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>

        <%@ include file="/Header.jsp"%>
        Current date :
        <%=new Date()%>

</body>
</html>
```

The output of JSP page.

(7) Best Practices of JSP

• **Don't overuse Java code in HTML pages**:

Putting all Java code directly in the JSP page is OK for very simple applications. But **overusing this feature leads to spaghetti code** that is not easy to read and understand. One way to minimize Java code in HTML pages is to **write separate Java classes** that **perform the computations**. Once these classes are tested, instances can be created.

• **Choose the right include mechanism**:

Static data such as **headers, footers, and navigation bar content is best kept in separate files** and not regenerated dynamically. Once such content is in separate files, they can be **included in all pages using one of the following include mechanisms**:

- Include directive: `<%@ include file="filename" %>`
- Include action: `<jsp:include page="page.jsp" flush="true" />`

The **first include** mechanism includes the content of the specified file while the JSP page is being converted to **a servlet (translation phase),** and the **second include** includes the **response generated after the specified page is executed**. Recommend using the **include directive**, which is fast in terms of

performance, if the file doesn't change often; and use the include action for content that changes often or if the page to be included cannot be decided until the main page is executed.

5.1.3 Understanding Servlet

Java Servlet technology and Java Server Pages (JSP pages) are server-side technologies the standard way to develop commercial web applications. Servlets and JSP pages help **separate presentation from content**. One best practice that **combines and integrates the use of servlets and JSP pages** is the **Model View Controller (MVC) design pattern.**

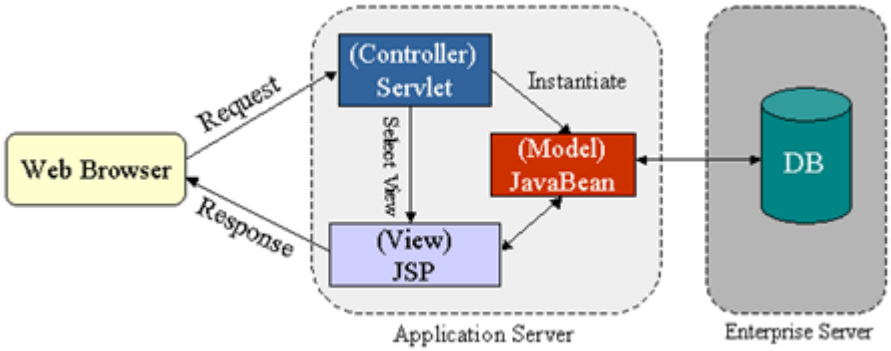


Figure 7.16: JSP and Servlet Integration Model Architecture

As shown in Figure 7.16, integrates the use of both servlets and JSP pages. In this mode, **JSP pages** are used for **the presentation layer**, and **servlets** for **processing tasks**. The **servlet acts as a controller responsible** for processing requests and creating any beans needed by the JSP page. The controller is also responsible for deciding to which JSP page to forward the request. The JSP page retrieves objects created by the servlet and extracts dynamic content for insertion within a template.

This model promotes the **use of the Model View Controller (MVC) architectural style design pattern**. Note that several frameworks already exist that implement this useful design pattern, and that truly separate presentation from content. The Apache Struts is a formalized framework for MVC. This framework is best used for complex applications where a single request or form submission can result in substantially different-looking results.

Servlet technology itself is used to create a web application (resides at server side and generates a dynamic web page). Servlet is an API that provides many interfaces and classes including documentation. **Servlets** support **a request and response programming model.** When a **client** sends **a request to the server**, the **server sends the request to the servlet**. The servlet then **constructs a response that the server sends back to the client**. However, **servlets run within the same process as the HTTP server**. Servlet is **a web component** that is deployed on the server **to create a dynamic web page**.

When a client request is made, the **service method** is called and passed **a request and response object.** The servlet first determines whether the request is a **GET** or **POST** operation. It then calls one of the following **methods**: **doGet() or doPost().** The **doGet()** method is called if the request is GET, and **doPost() method** is called if the request is POST. Both doGet() and doPost() methods take request (HttpServletRequest) and response (HttpServletResponse).

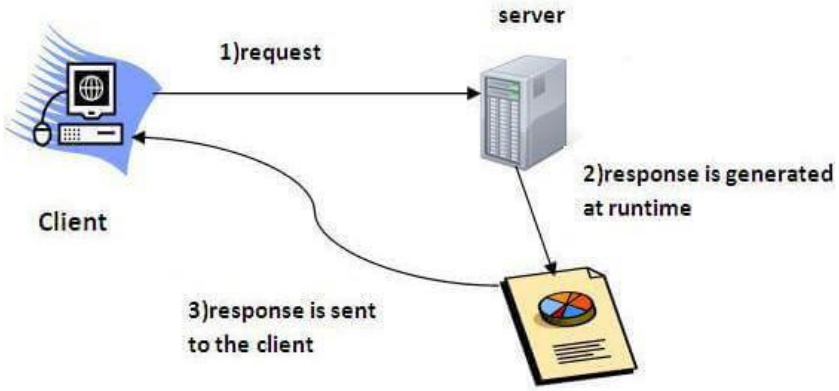


Figure 7.17: Servlets' *Request/Response flow calling a JSP page*

In the simplest terms, then, servlets are **Java classes** that **can generate dynamic HTML content using print statements**. Servlets, however, run in a container, and the APIs provide session and object life-cycle management. Consequently, when you use servlets, you gain all the benefits from the Java platform, which include the sandbox (security), database access API via JDBC, and cross-platform portability of servlets.

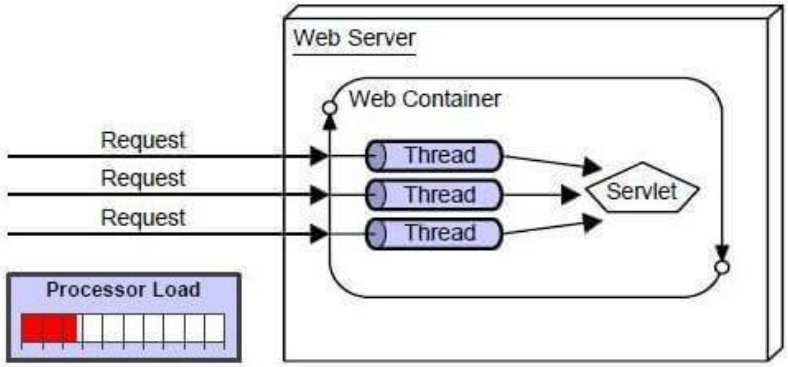


Figure 7.18: Servlet runs in a container and handle requests as threads

Servlet Example

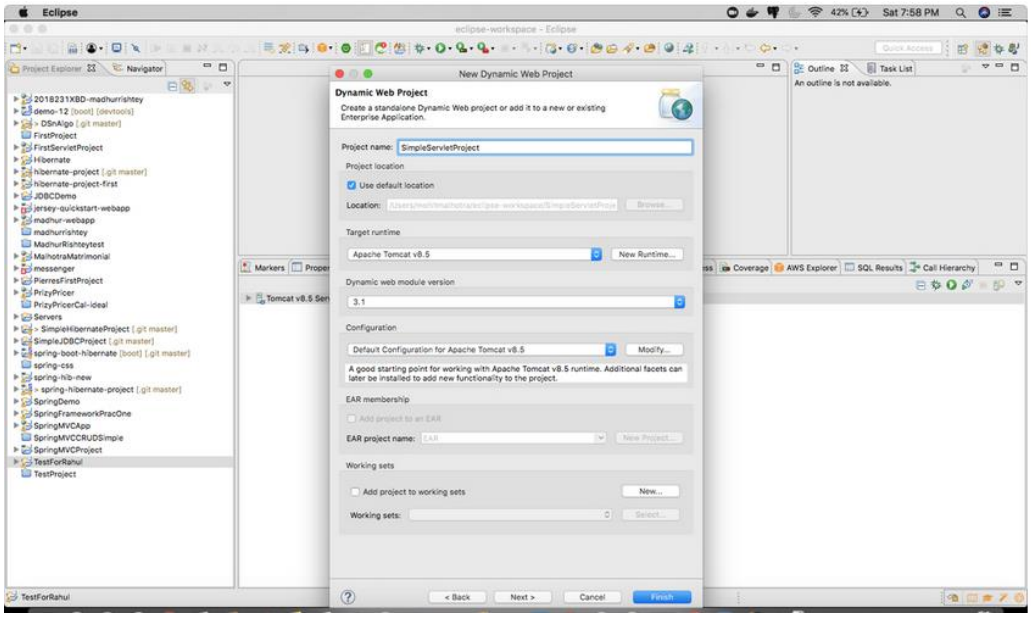
Let's create a simple servlet program that displays "Hello World".

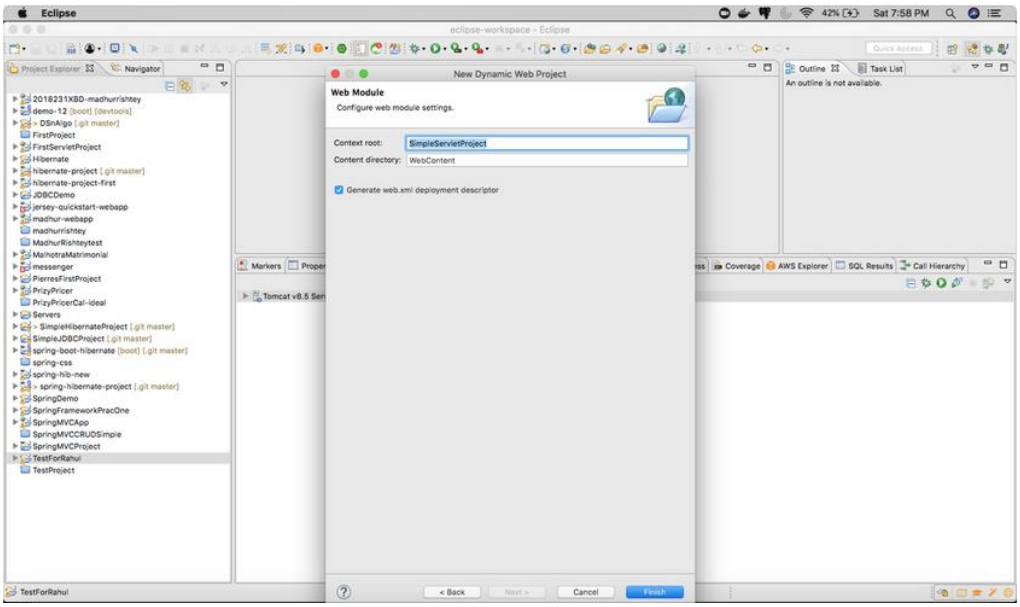
Step 1: Create a new Dynamic web project – "SimpleServletProject"

Context root: SimpleServletProject

Context directory: WebContent

Check the "Generate web.xml deployment descriptor and Finish





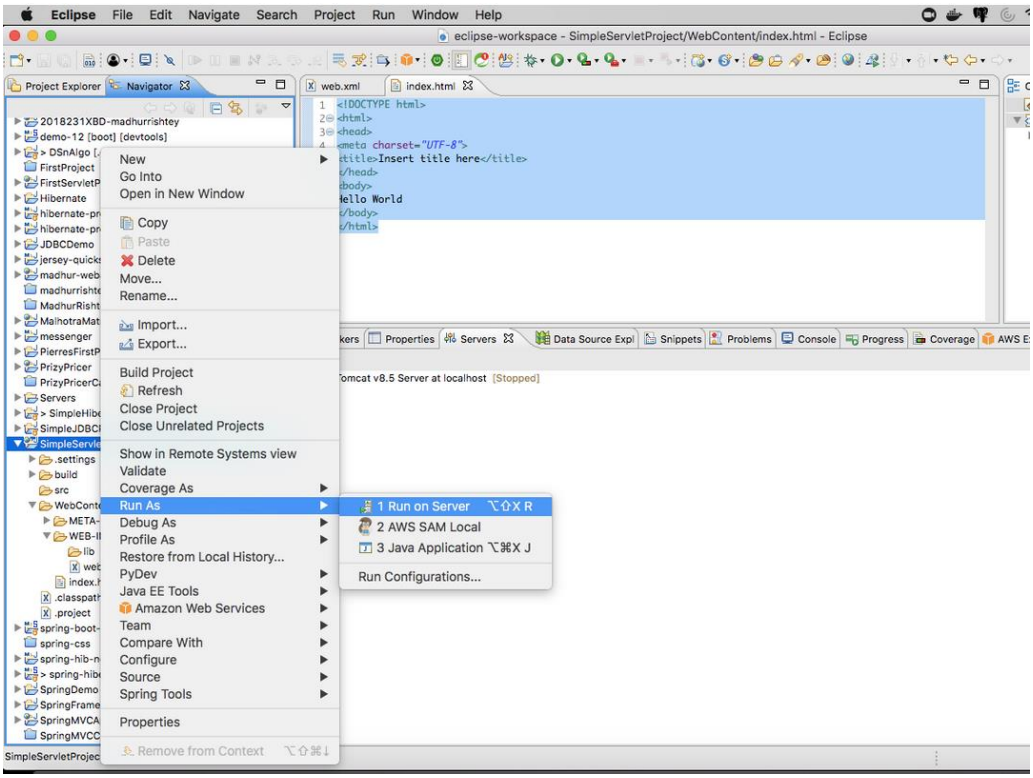
Step 2: Open web.xml

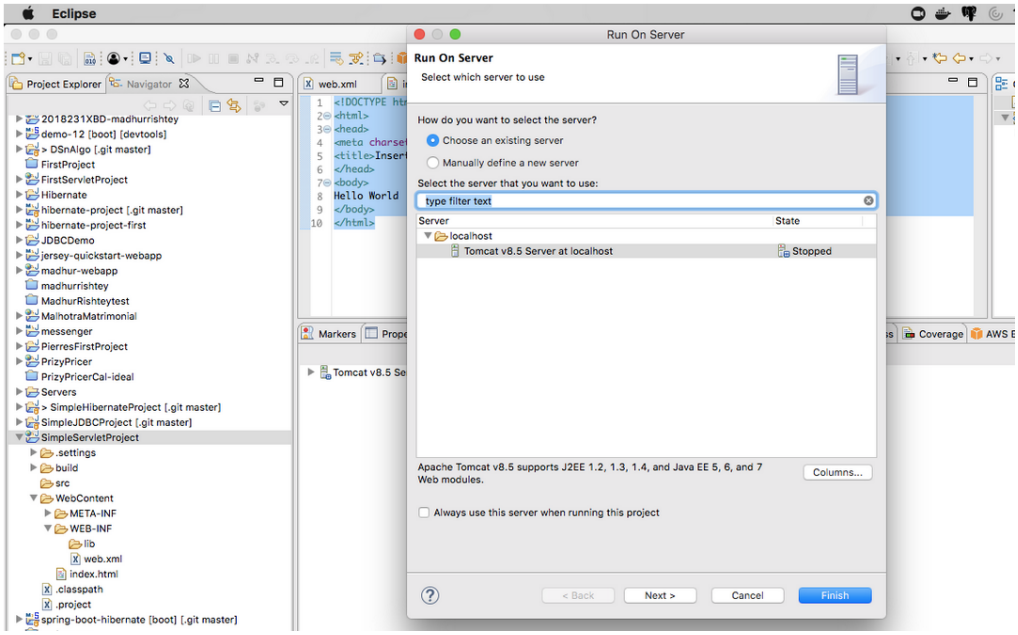
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" id="WebApp_ID"
version="3.1">
  <display-name>SimpleServletProject</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

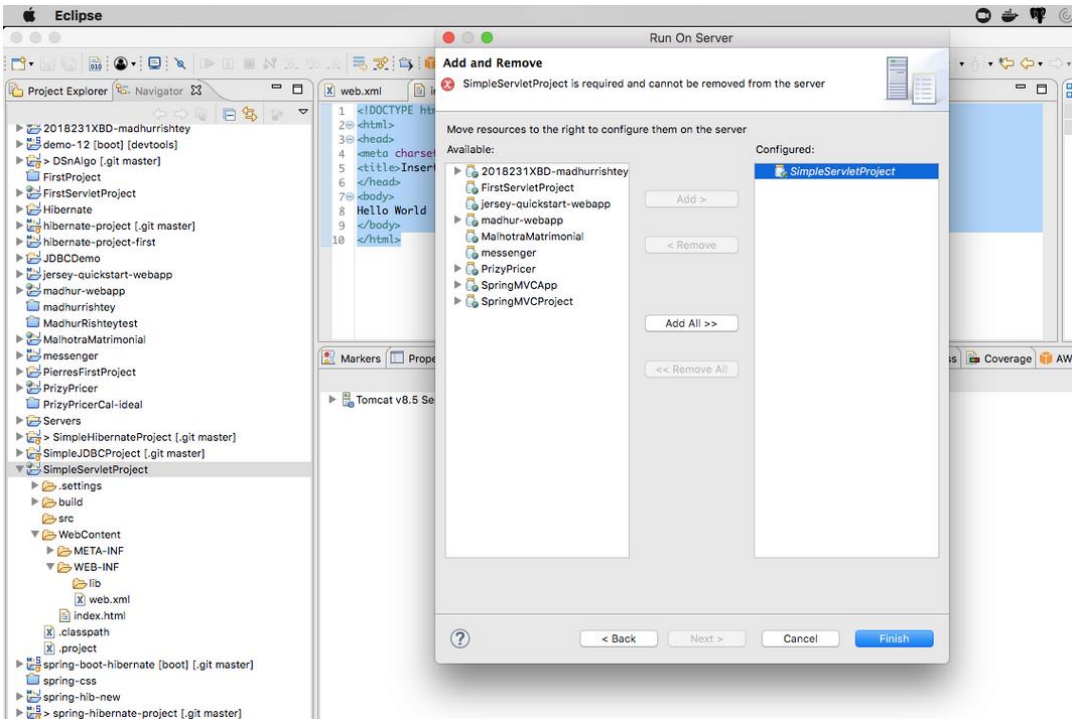
Step 3: Create a new html file – “index.html”

```
<!DOCTYPE html>
<html>
<head>
<meta charset=”UTF-8”>
<title>Insert title here</title>
</head>
<body> Hello World </body>
</html>
```

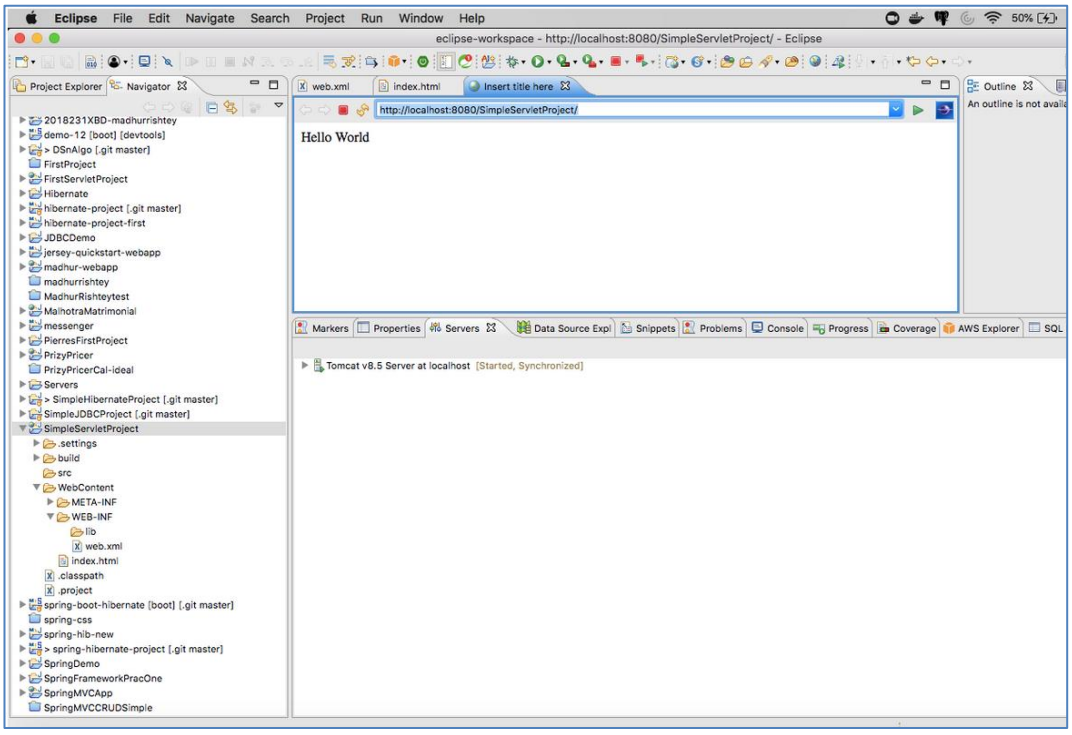
Step 4: Run on server



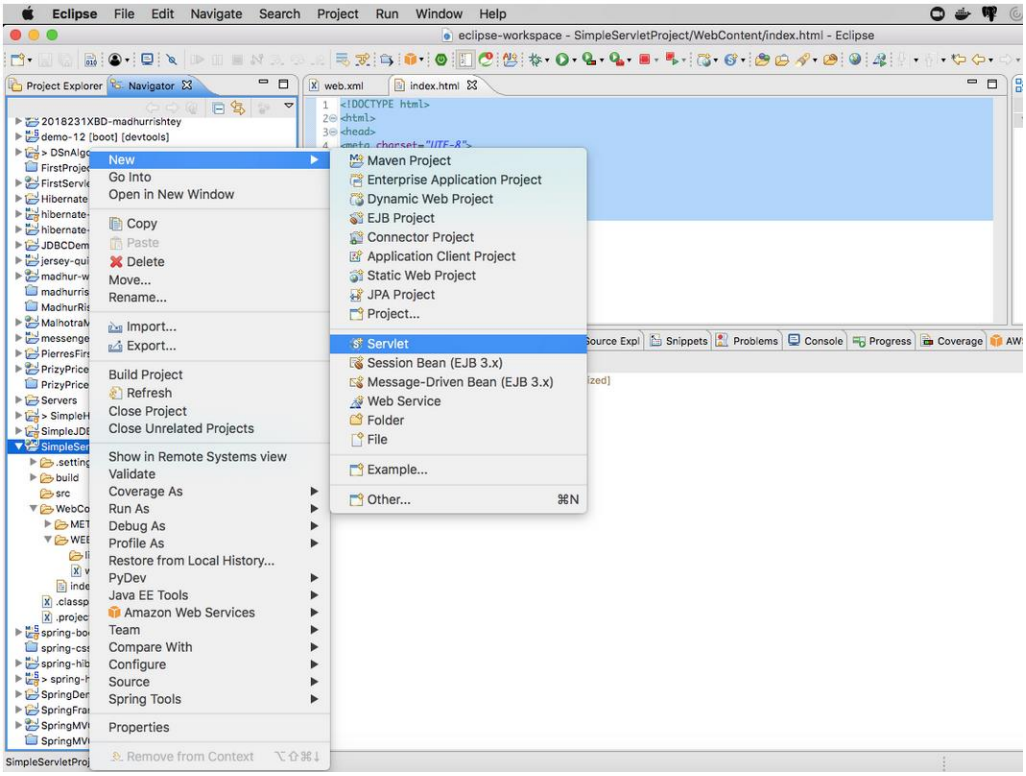


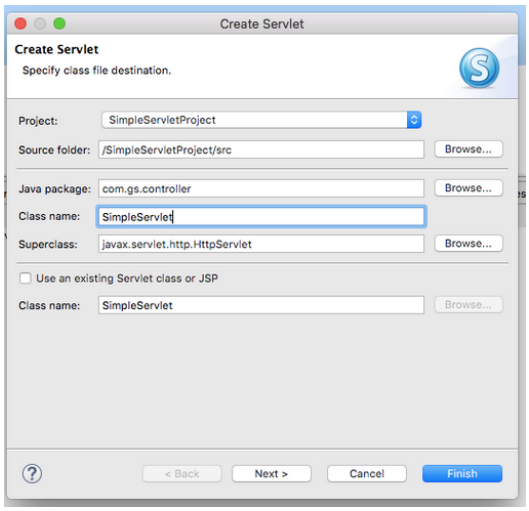


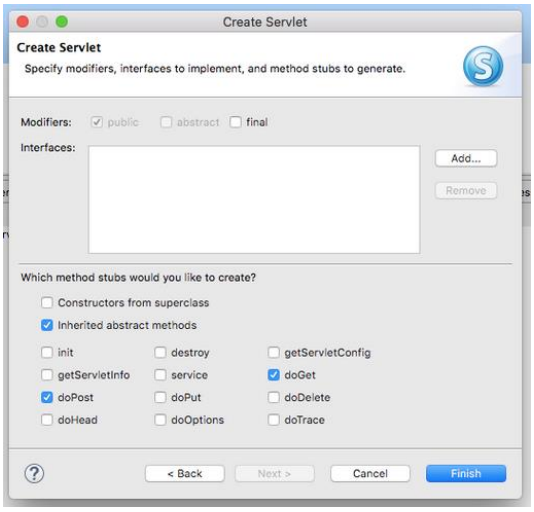
Step 5: Validate output on browser



Step 6: Create a new servlet – “SimpleServlet.java”

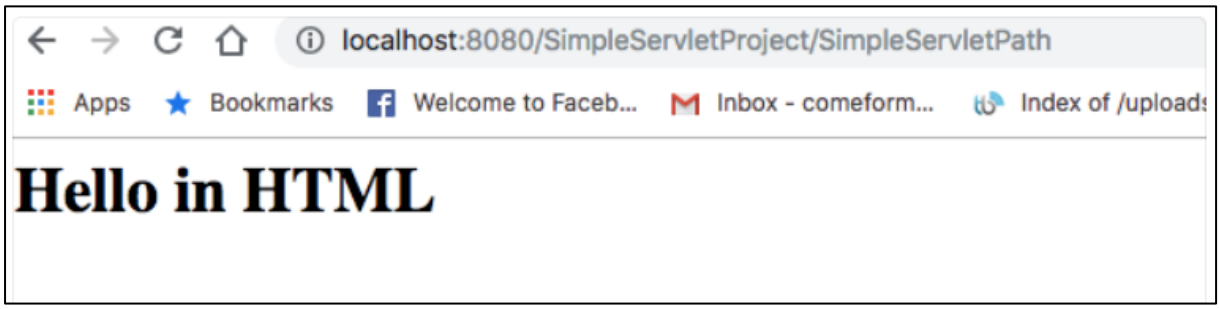






```
package com.gs.controller;

import java.io.IOException;
import java.io.PrintWriter;

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

/**
 * Servlet implementation class SimpleServlet
 */
@WebServlet("/SimpleServletPath")
public class SimpleServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;

	/**
	 * @see HttpServlet#doGet(HttpServletRequest request,
HttpServletResponse response)
	 */
	protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
		System.out.println("Hello from Get method");
		PrintWriter printWriter =  response.getWriter();
		printWriter.println("<h1>Hello in HTML</h1>");
	}

	/**
	 * @see HttpServlet#doPost(HttpServletRequest request,
HttpServletResponse response)
	 */
	protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
		// TODO Auto-generated method stub
		doGet(request, response);
	}

}
```

Step 7: Run on server and output on the browser



Mapping Servlet Path

There are two ways to map servlet path:

(1) Using Annotation "@WebServlet (….)" in Servlet class

Example: `@WebServlet("/SimpleServletPath")`

See code snipped from `SimpleServlet`.java

```
package com.gs.controller;

import java.io.IOException;
import java.io.PrintWriter;…

/** * Servlet implementation class SimpleServlet  */
@WebServlet("/SimpleServletPath")

public class SimpleServlet extends HttpServlet {
 private static final long serialVersionUID = 1L;
    //code not shown
    ...
  protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
{
      …
 }
 …
}
```

@WebServlet

`@WebServlet` defines a servlet component and its metadata, with the following attributes:

- `String[] urlPatterns`: An array of String declaring the `url-pattern` for `servlet-mapping`. Default is an empty array `{}`.
- `String[] value`: `urlPatterns`.
- `String name`: `servlet-name`, default is empty string `""`.
- `loadOnStartup`: The `load-on-startup` order of the servlet, default is -1.
- `WebInitParam[] initParams`: The init parameters of the servlet, default is an empty array {}.
- `boolean asyncSupported`: Declares whether the servlet supports asynchronous operation mode, default is false.
- `String smallIcon`, `String largeIcon`, `String description`: icon and description of the servlet.

Example:

```
@WebServlet("/sayHello")
public class Hello1Servlet extends HttpServlet { ..... }
  // One URL pattern

@WebServlet(urlPatterns = {"/sayhello", "/sayhi"})
public class Hello2Servlet extends HttpServlet { ..... }
  // More than one URL patterns
```

@WebInitParam

`@WebInitParam` is Used to declare init params in servlet, with the following attributes:

- `String name` and `String value` (required): Declare the name and value of the init parameter.
- `String description` (optional) description, default empty string `""`.

(2) Using web.xml

Example:

```
<servlet>
  <servlet-name> … </servlet-name>
  <servlet-class> … </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name> … </servlet-name>
  <url-pattern> /… </url-pattern>
</servlet-mapping>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
 http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
 id="WebApp_ID" version="3.1">
  <display-name>SimpleServletProject</display-name>
  <servlet>
     <servlet-name>simpleServlet</servlet-name>
     <servlet-class>
        com.gs.controller.SimpleServlet</servlet-class>
  </servlet>
  <servlet-mapping>
     <servlet-name>simpleServlet</servlet-name>
     <url-pattern>/SimpleServletPath</url-pattern>
  </servlet-mapping>
</web-app>
```

Understanding GET and POST methods in Servlet

Step 1: Passing data using parameters in GET method:

```
package com.gs.controller;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class XmlServlet extends HttpServlet {

        protected void doGet(HttpServletRequest request,
HttpServletResponse response)throws ServletException, IOException {
        System.out.println("Hello from Get method of XML servlet");
        response.setContentType("text/html");
        String userName =  request.getParameter("userName");
        PrintWriter printWriter = response.getWriter();
        printWriter.println("Hello from GET method "+userName);

        }
}
```

Step 2: Hit the URL and add the parameter as "?userName=Mohit"

```
http://localhost:8080/SimpleServletProject/xmlservletpath?userName=Mohit
```

Step 3: In order to test POST create a simple html form: "index.html"

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
        Hello World
        <form method="post" action="xmlservletpath">
                <input name="userName" />
                <input type="submit" />
        </form>
</body>
</html>
```

Step 4: Create doPost method in servlet:

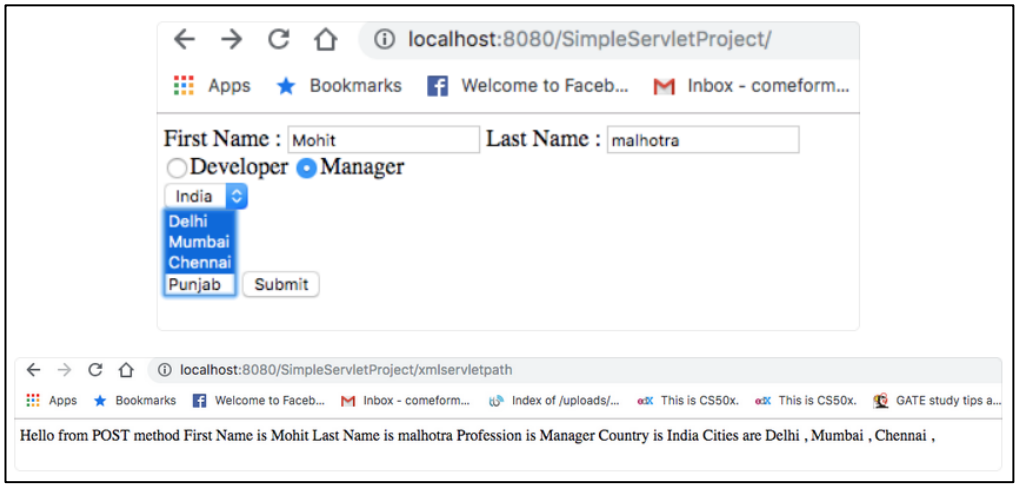
```
package com.gs.controller;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class XmlServlet extends HttpServlet {

  protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        System.out.println("Hello from Get method of XML servlet");
        response.setContentType("text/html");
        String userName = request.getParameter("userName");
        PrintWriter printWriter = response.getWriter();
        printWriter.println("Hello from GET method " + userName);
}

  protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        System.out.println("Hello from POST method of XML servlet");
        response.setContentType("text/html");
        String userName = request.getParameter("userName");
        PrintWriter printWriter = response.getWriter();
        printWriter.println("Hello from POST method " + userName);
        }
}
```

Step 5: Map the servlet class in web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
 http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
 id="WebApp_ID" version="3.1">

  <display-name>SimpleServletProject</display-name>
  <servlet>
     <servlet-name>xmlServlet</servlet-name>
     <servlet-class>com.gs.controller.XmlServlet</servlet-class>
  </servlet>
  <servlet-mapping>
     <servlet-name>xmlServlet</servlet-name>
     <url-pattern>/xmlservletpath</url-pattern>
  </servlet-mapping></web-app>
</web-app>
```

Output

Passing more parameters: How to get values from radio and select?



Step 1 : change index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
        <form method="post" action="xmlservletpath">
            First Name : <input name="userName" />
            Last Name :  <input name="lastName" />
            <br>
```

```
            <input type="radio" name="profession" value =
"Developer"/>Developer
            <input type="radio" name="profession" value =
"Manager"/>Manager
            <br>
            <select name = "countries">
                  <option value="India">India</option>
                  <option value="Pak">Pak</option>
                  <option value="USA">USA</option>
            </select>
            <br>
            <select name = "cities" multiple="3">
                  <option value="Delhi">Delhi</option>
                  <option value="Mumbai">Mumbai</option>
                  <option value="Chennai">Chennai</option>
                  <option value="Delhi">Punjab</option>
                  <option value="Mumbai">Agra</option>
                  <option value="Chennai">Mathura</option>
            </select>

            <input type="submit" />
    </form>
</body>
</html>
```

Step 2: Change the controller (XmlServlet.java):

To get multiple values from select option, see code snipped in **doPost**() method of the **XmlServlet.java**..

```
String[] cities = request.getParameterValues("cities");
package com.gs.controller;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class XmlServlet extends HttpServlet {

        protected void doGet(HttpServletRequest request,
HttpServletResponse response)throws ServletException, IOException {
        System.out.println("Hello from Get method of XML servlet");
        response.setContentType("text/html");
        String userName = request.getParameter("userName");
        PrintWriter printWriter = response.getWriter();
        printWriter.println("Hello from GET method " + userName);
        }

        protected void doPost(HttpServletRequest request,
HttpServletResponse response)throws ServletException, IOException {
        System.out.println("Hello from POST method of XML servlet");
        response.setContentType("text/html");
        String userName = request.getParameter("userName");
        String lastName = request.getParameter("lastName");
        String profession = request.getParameter("profession");
        String countries = request.getParameter("countries");
        String[] cities = request.getParameterValues("cities");
        PrintWriter printWriter = response.getWriter();
        printWriter.println("Hello from POST method \n");
        printWriter.println("First Name is \n " + userName);
        printWriter.println("Last Name is \n" + lastName);
```

```
        printWriter.println("Profession is \n " + profession);
        printWriter.println("Country is \n " + countries);
        printWriter.println("Cities are \n ");
        for (String s : cities) {
                printWriter.println(s+" ,");
        }
    }
}
```

Request, Session and Context Objects

(1) Session Object:

1. One per user/machine. Session object is null for other user/browser.
2. Objects available across requests
3. Perfect for login sessions and shopping carts
4. Every request object has a reference to the session object.

(2) Context Object:

1. Across the entire application
2. Shared across servlets and users
3. Initialization code / Common bulletin board
4. Use the Context object

(3) How to create session?

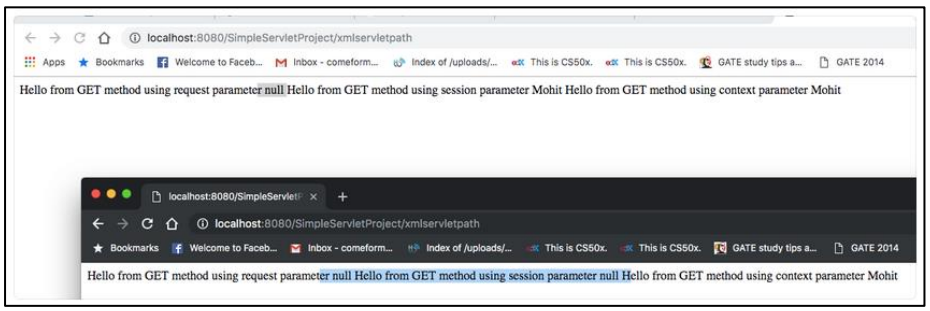
Code snippet from doGet() method in XmlServlet.java to get parameter passing value and set it as **session attribute value to set value for one per user/machine** and **context attribute value to share across the entire application:**

```
        //getting session
        HttpSession httpSession = request.getSession();

        //getting context
        ServletContext context = request.getServletContext();
        if (userName != null) {
                httpSession.setAttribute("userName", userName);
                context.setAttribute("userName", userName);
        }
```

```
ServletContext context = request.getServletContext();
  if (userName != null) {
        httpSession.setAttribute("userName", userName);
        context.setAttribute("userName", userName);
  }
protected void doGet(HttpServletRequest request, HttpServletResponse
response)       throws ServletException, IOException {
        System.out.println("Hello from Get method of XML servlet");
        response.setContentType("text/html");
        String userName = request.getParameter("userName");
        PrintWriter printWriter = response.getWriter();
        printWriter.println("Hello from GET method using request parameter
" + userName);
        //getting session
        HttpSession httpSession = request.getSession();
        //getting context
        ServletContext context = request.getServletContext();
        if (userName != null) {
            httpSession.setAttribute("userName", userName);
            context.setAttribute("userName", userName);
```

```
        }
        printWriter.println("Hello from GET method using session parameter
" + httpSession.getAttribute("userName"));
        printWriter.println("Hello from GET method using context parameter
" + context.getAttribute("userName"));
}
```

Output:

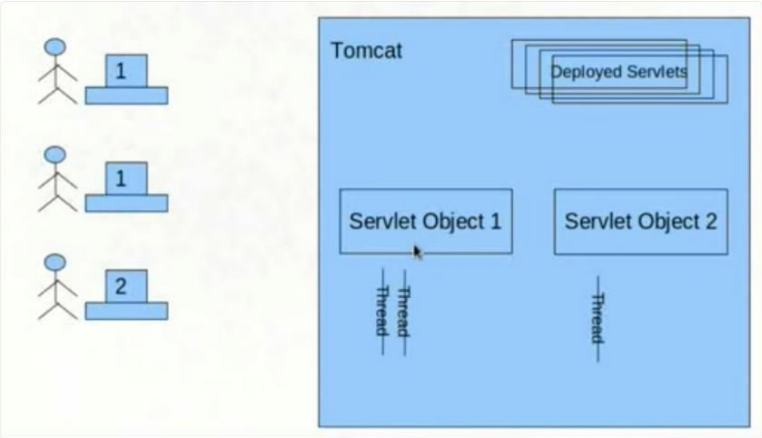


Understanding init(), service() and ServletConfig()

Servlet objects are created in two phases:

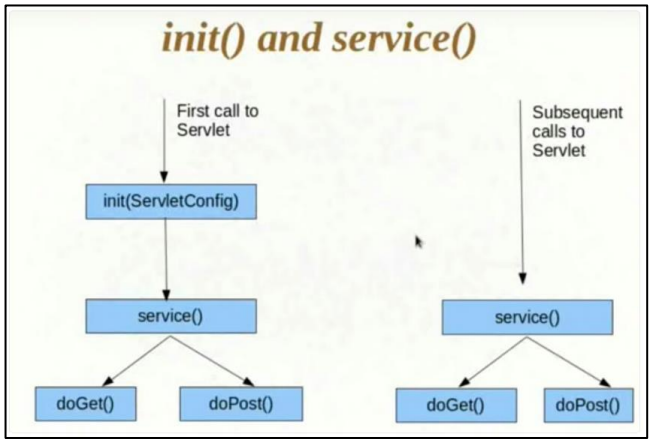
1) At the first call

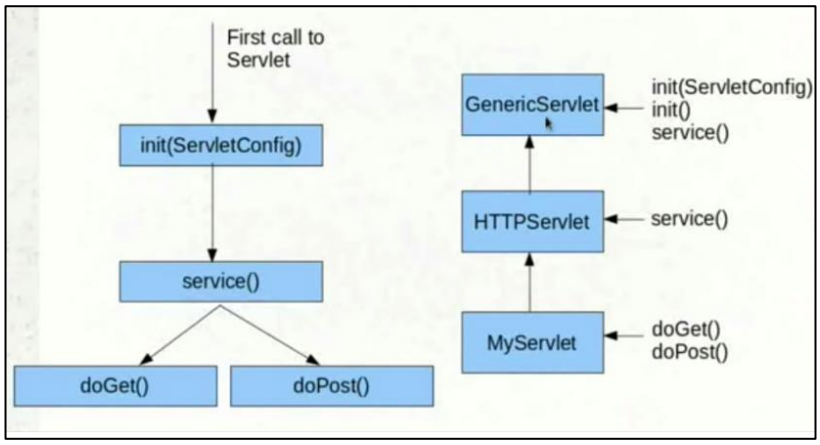
2) Next only threads are created for each request.



There are methods which run before doGet() or doPost () method, they are as follows:

1. **init()**
2. **service()**





init() and init(ServletConfig) and service are part of GenericServlet

```
@Override
public void init(ServletConfig config) throws ServletException {
     this.config = config;
     this.init();
}
```

service() (overidden from GenericServlet) and doGet ..... are part of HttpServlet

```
@Override
public void service(ServletRequest req, ServletResponse res)
         throws ServletException, IOException{
    HttpServletRequest request;
    HttpServletResponse response;

    try{
         request = (HttpServletRequest) req;
         response = (HttpServletResponse) res;
    } catch(ClassCastException e){
    throw new ServletException("non-HTTP request or response");
    }
    Service(request, response);
  }
}
```

Examples:

1) **Initital parameter using initparams Annotation**

```
package com.gs.controller;

import java.io.IOException;
import java.io.PrintWriter;

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

/**
 * Servlet implementation class SimpleServlet
 */
@WebServlet(description = "Simple hello", urlPatterns = {
"/SimpleServletPath" }, initParams = {@WebInitParam(name = "defaultUser",
value = "Mohit Malhotra") })
```

```
public class SimpleServlet extends HttpServlet {
        private static final long serialVersionUID = 1L;

/**
  * @see HttpServlet#doGet(HttpServletRequest request,
HttpServletResponse  response)
  */
  protected void doGet(HttpServletRequest request, HttpServletResponse
response)
     throws ServletException, IOException {
        System.out.println("Hello from Get method");
        PrintWriter printWriter = response.getWriter();
        printWriter.println("<h1>Hello in HTML</h1>");
        String initParamValue =
this.getServletConfig().getInitParameter("defaultUser");
        printWriter.println("Hello from init method " + initParamValue);
  }

  /**
    * @see HttpServlet#doPost(HttpServletRequest request,
HttpServletResponse response)
    */
  protected void doPost(HttpServletRequest request, HttpServletResponse
response)
     throws ServletException, IOException {
        // TODO Auto-generated method stub
                doGet(request, response);
        }
}
```

Output

2) Initial parameter using web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://xmlns.jcp.org/xml/ns/javaee"
      xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
      id="WebApp_ID" version="3.1">
  <display-name>SimpleServletProject</display-name>
  <servlet>
      <servlet-name>xmlServlet</servlet-name>
      <servlet-class>com.gs.controller.XmlServlet</servlet-class>
      <init-param>
            <param-name>defaultUser</param-name>
            <param-value>Mohit Malhotra from XML</param-value>
      </init-param>
      </servlet>
      <servlet-mapping>
            <servlet-name>xmlServlet</servlet-name>
            <url-pattern>/xmlservletpath</url-pattern>
      </servlet-mapping>
</web-app>
```

Code snippet : XmlServlet.java

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class XmlServlet extends HttpServlet {

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

      System.out.println("----Hello from Get method of XML servlet----");
      response.setContentType("text/html");
      String userName = request.getParameter("userName");
      PrintWriter printWriter = response.getWriter();
      printWriter.println("---Hello from GET method using request
parameter--- " + userName);
      //getting session
      HttpSession httpSession = request.getSession();
      //getting context
      ServletContext context = request.getServletContext();
      if (userName != null) {
            httpSession.setAttribute("userName", userName);
            context.setAttribute("userName", userName);
      }
      printWriter.println("---Hello from GET method using session
parameter--- " + httpSession.getAttribute("userName"));
      printWriter.println("---Hello from GET method using context
parameter--- " + context.getAttribute("userName"));
      String initParamValue =
this.getServletConfig().getInitParameter("defaultUser");
      printWriter.println("---Hello from init method using context
parameter--- " + initParamValue);
      }
}
```

Output

5.2 Map the Classes/Objects to Relational Tables

In our Library System which we had developed in previous lecture-3; the system includes classes such as Library, Member, Book, Catalog, and so on. The class diagram is shown in Figure 7.19.

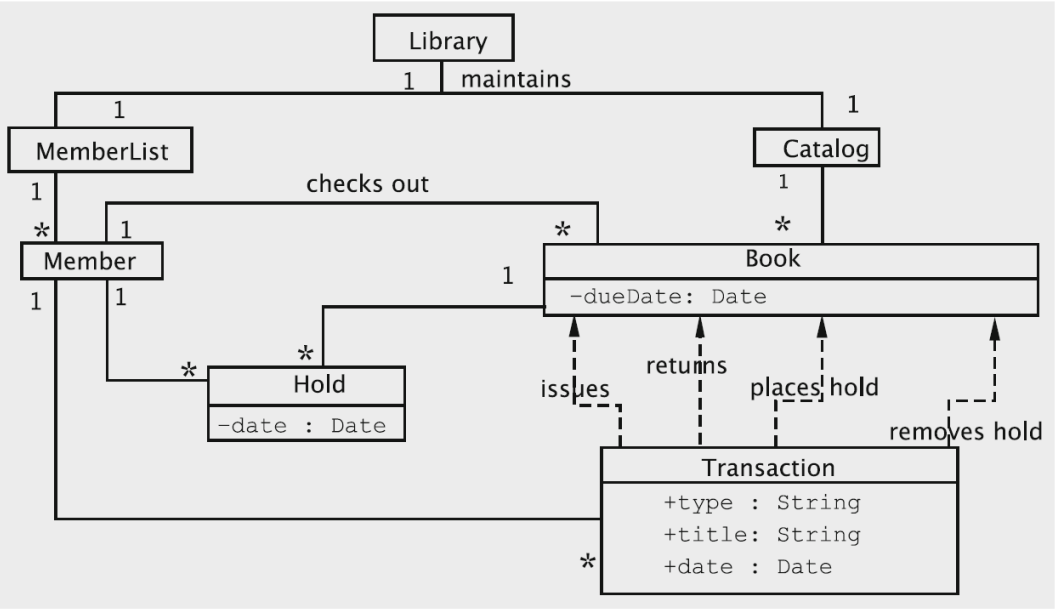


Figure 7.19: Class diagram for the Library Automation System (lecture 3)

Mapping these classes to tables based on their relations, we can construct the relational tables as in the follow section. First consider the `Library` class, which has only two attributes; `MemberList` and `Catalog`. Since these two attributes are collections of `Member` objects and `Book` objects, so we only need to store `Member` and Book objects separately. So we do not need to map the `Library` class to tables.

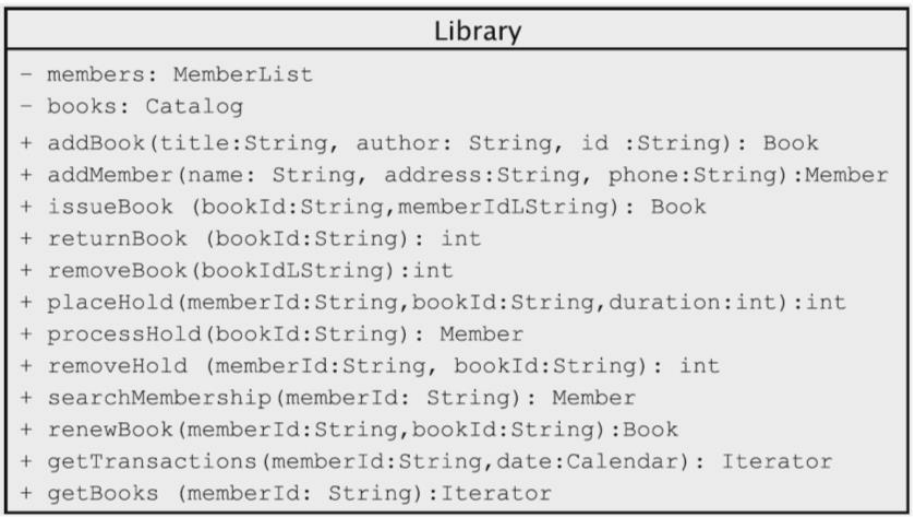


Figure 7.20: Class diagram for Library class

Similarly the MemberList class and Catalog class is the collection class for Member objects and Book objects. We do not need to store them in tables.

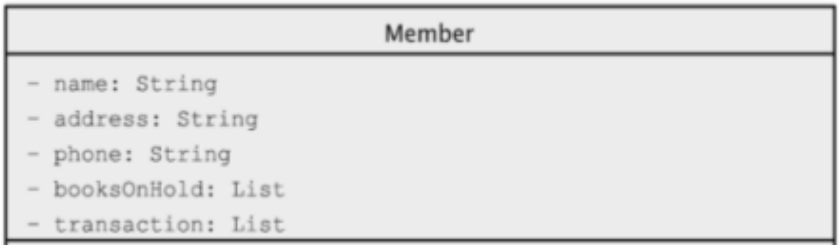


Figure 7.21: Class diagram for Member class

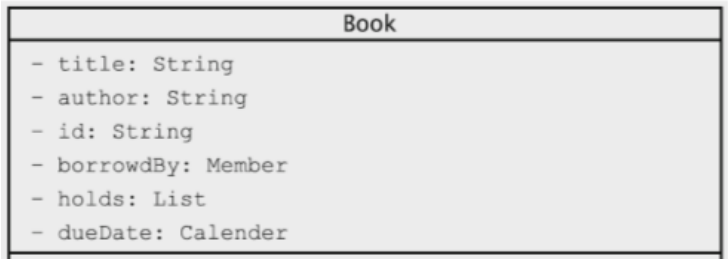


Figure 7.22: Class diagram for Book class

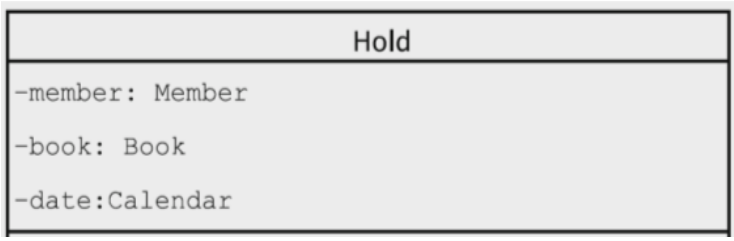


Figure 7.23: Class diagram for Hold class

For Member class, the mapping to tables as follows. We also need to add password attributes to login to the system. According to requirement statement, Super-users have special ***User IDs*** and corresponding ***password***. For regular members, their library ***Member ID*** will be their user id and **their phone number** will be the ***password***.

According to the one-to-many association between Member and Book class with Hold as an association class, we will use second method to store in then in separate tables. The attribute booksOnHold is a collection list of books objects and we use ID field to connect between Member and Book and Hold.

Member Table

Attribute Name	**Null ?**	**Domain**
MemberID	Not Null	ID
Name	Not Null	Name
address	Not Null	Address
Phoneno	Not Null	Phone Number
Password	Not Null	Login password

Book Table

Attribute Name	**Null ?**	**Domain**
BookID	Not Null	ID
Title	Not Null	Book Title
Author	Not Null	Book Author Name
borrowedBy	Null	Member ID
dueDate	Null	Book Return Date

Hold Table

Attribute Name	**Null ?**	**Domain**
MemberID	Not Null	MemberID
BookID	Not Null	Book ID
ExpireDate	Not Null	Duration of Hold Date

For Super-users, the Staff table keeps their special ***User IDs*** and corresponding ***password***, name, rank, phone number.

Staff Table

Attribute Name	**Null ?**	**Domain**

StaffID	Not Null	ID
Name	Not Null	Name
Rank	Not Null	Job Title
Phoneno	Not Null	Phone Number
Password	Not Null	Login password

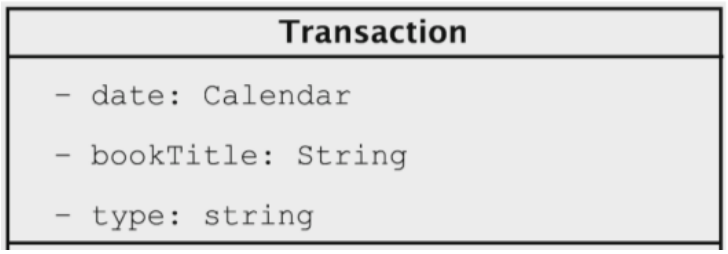


Figure 7.24: Class diagram for Transaction

The one-to-many association between Member and Book class with Transaction class, map to the Transaction table as follows. Keeping the BookID can refer to the book title so we do not keep the book title in the Transaction table.

Transaction Table

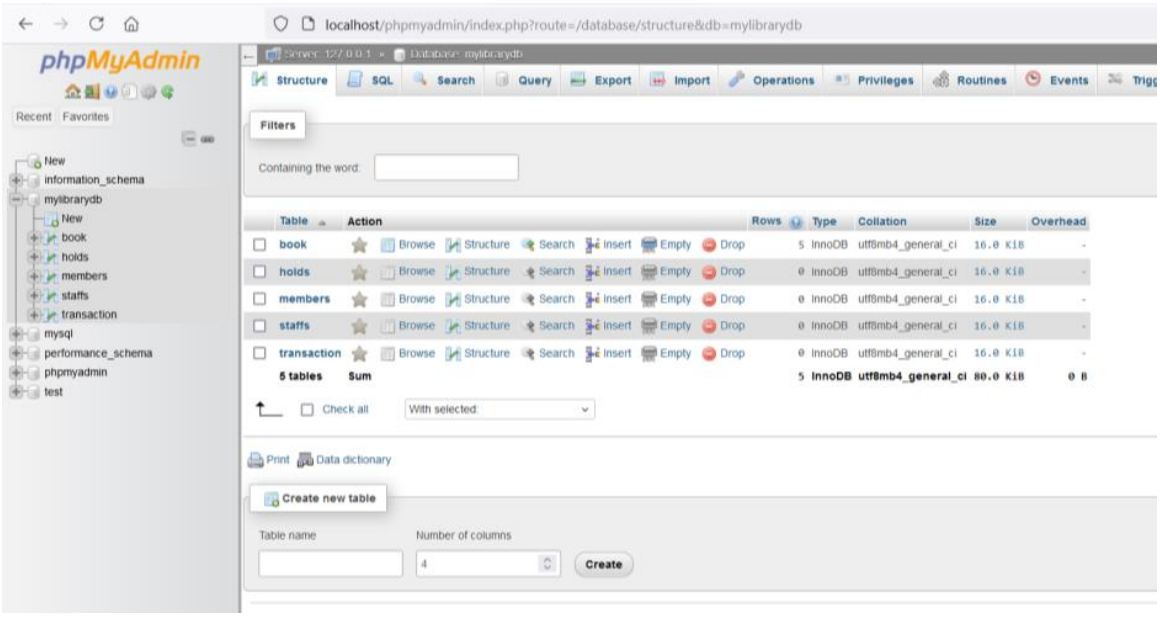
Attribute Name	**Null ?**	**Domain**
TransID	Not Null	TransID – Auto increment
MemberID	Not Null	MemberID
StaffID	Not Null	StaffID
BookID	Not Null	Book ID
TransDate	Not Null	Transaction Date
TransType	Not Null	Issue, Return, Place Hold, Remove Hold, Add book/ member, Remove book, process hold

5.2.1 Database Creation

In the web-based system, the permanent data (created by the save command) that stores information about the members, books, who borrowed what, holds, etc., are saved to the database. After mapping the Class diagrams to relational tables, we will build tables as follows. We will use XAMPP - MySQL to store data.

(1) Creating Database using phpMyAdmin

- Open the XAMPP control and start the Apache and MySQL server
- Go to your browser and write localhost/dashboard >> phpmyadmin
- Now you can create your own database by using New button.
- Create a database named “librarydb” and inside it create separate databases as follows:



Database Used in this project:

Below is the SQL code to create those tables in the database, You can modify the code to create your own database for the project.

```
CREATE TABLE `staffs` (
  `staffid`  varchar(10) NOT NULL,
  `name`     varchar(100) NOT NULL,
  `rank`     varchar(100) NOT NULL,
  `phoneno`  int(10) NOT NULL,
  `password` varchar(10) NOT NULL,
   PRIMARY KEY (`staffid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

--
-- Dumping data for table `staffs`
--

INSERT INTO `staffs` ( 'staffid', `name`, `rank`, `phoneno`, `password`) VALUES
('admin01', 'admin', 'Manager', 094438757, 'steve@123');
```

```
-- --------------------------------------------------------
--
-- Table structure for table `books`
--

CREATE TABLE `books` (
  `bookid`  varchar(25) NOT NULL,
  `title`   varchar(250) NOT NULL,
  `author`     varchar(250) NOT NULL,
  `borrowedby` varchar(10)  NULL,
  `duedate`    varchar(10)  NULL,
   PRIMARY KEY (`bookid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

--
-- Dumping data for table `books`
--
```

```
INSERT INTO `books` (`bookid`, `title`, `author`) VALUES
('SW1', 'Software engineering', 'M D Guptaa'),
('SW2', 'Data structure', 'Munshi Prem Chand'),
('JAVA1', 'Java Implementation', 'M D Guptaa'),
('SW3', 'Object Oriented Software Design', 'Munshi Prem Chand'),
('MY1', 'The Guest Book, Mystery', 'Munshi Prem Chand'),
('FIC1', 'The Perfect Murder, Fiction', 'Munshi Prem Chand'),
('BIO1', 'Accidential Presidents, Biography', 'Munshi Prem Chand'),
('MY2', 'War and Peace, Mystery', 'John Smith'),
('FIC2', 'The Wicked King, Fiction', 'Brown' )
('THRIL1', 'Murder on the Orient Express, Thriller', 'Henry');

-- --------------------------------------------------------
--
-- Table structure for table `ordinary members`
--

CREATE TABLE `members` (
  `memberid`  varchar(10) NOT NULL,
  `name`      varchar(50) NOT NULL,
  `phoneno`   int(10) NOT NULL,
  `password`  varchar(10) NOT NULL,
  PRIMARY KEY (`memberid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

--
-- Dumping data for table `members`
--

INSERT INTO `members` (`memberid`, `name`, `phoneno`,`password` ) VALUES
((‘mem01’,'Mg Mg', 095449920, mgmg@123),
(‘mem02’, 'Ma Ma', 091223451, mama@123),
(‘mem03’, 'Ko Ko', 096659090, koko@123));

-- --------------------------------------------------------
--
-- Table structure for table `transactions`
--

CREATE TABLE `transactions` (
  `transid`   int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=1;
  ‘memberid’  varchar(10) NOT NULL,
  `staffid`   varchar(10) NOT NULL,
  `bookid`    varchar(25) NOT NULL,
  `transtype` varchar(20) NOT NULL,
  `transdate` varchar(10) NOT NULL,
   PRIMARY KEY (`transid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

-- --------------------------------------------------------
--
-- Table structure for table `holds`
--

CREATE TABLE `holds` (
  `memberid` varchar(10) NOT NULL,
  `bookid` varchar(25) NOT NULL,
  `expiredate` varchar(10) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

After creating the database we can now start building the **frontend and backend of our project.**

5.3 Front End and Back End Development for Our Project

We now undertake the task of implementing a web-based version of the library system. We are going to develop our front end and back end part of the project in different modules. Here start with the **login process using JSP and Servlet technology** in the following section.

5.3.1 Initialization of Library Class

In our design, there is only one instance of `Library` class for each user. We need to initialize the data from relational tables to object classes. For example, we need to load staffs, members and books data from database into their classes; `StaffList`, `MemberList` and `Catalog` whenever the `Library system` is loaded. We use only one **`LibraryServlet.`**`java` **class** and all other servlets classes such as `loginservlet`, `addbookservlet`, `addmemberservlet`, etc. inherit from the `LibraryServlet` class which has `run()` **abstract method** to be implemented by these inherited servlet classes.

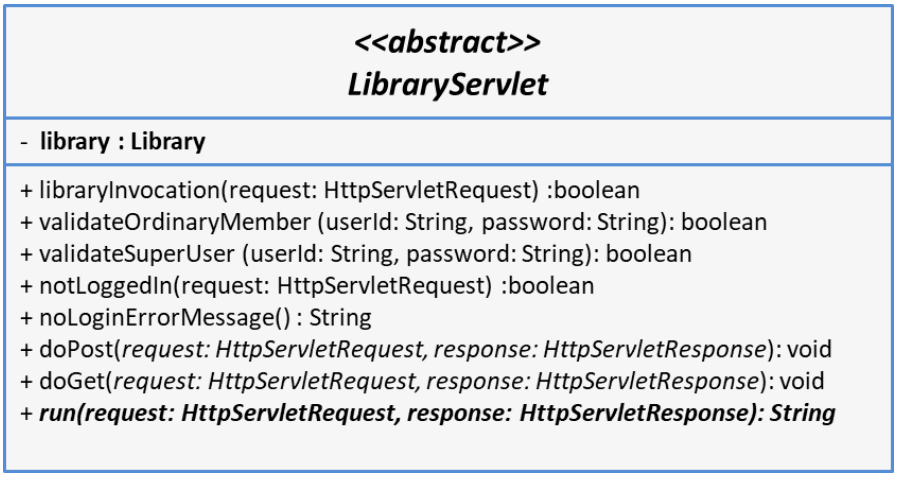


Figure 7.25: Class diagram for Library servlet

The `validateSuperUser()` method checks whether the user is a superuser and `validateOrdinaryMember()` check for ordinary members and help assess rights. The method `libraryInvocation` returns true if and only if the user has logged in from a terminal located within the library. The method `notLoggedIn()` returns true if and only if the user has not currently logged in. The method `noLoginErrorMessage()` returns HTML code that displays an error message when a person who has not logged in attempts to execute a command. The `doGet` message calls `doPost`, which does some minimal processing needed for all commands and then calls the abstract `run` method, which individual servlets override.

5.3.2 Role-based Login

Here start with the **Login page** as discussed in the requirement of **Logging in and the Initial Menu,** the login process is **user's role based**; the super user and the library members. After user has successfully login in, the initial menu will be displayed as follows:

(1) Super-user menu includes

(1) Adding New Books
(2) Returning Books
(3) Removing Book

(4) Adding New Member
(5) Processing Holds

(2) Member menu includes

(1) Issue Books
(2) Place Hold
(3) Remove Hold
(4) Print Transactions
(5) Renew Book

According to the MVC pattern, we will create the following files for login process;

- **View - Login.jsp, AdminHome.jsp, MemberHome.jsp**
  - The Login.jsp contains a simple HTML form to key in login credentials.
  - The AdminHome.jsp contains a Super user home page after successful log-in
  - The MemberHome.jsp contains a Member home page based on their roles of super user or member.

- **Controller – LibraryServlet.java, LoginServlet.java**
  - LibraryServlet.java is only one servlet file which acts as container to accepts multiple request from browser and Login details are forwarded to LoginServlet.java
  - LoginServlet.java class is the control class that will call necessary method from Library.java and it contains business logic and process HTTP request parameters and redirect to the appropriate JSP page (Admin or Member Home).

- **Model –LoginBean.java, DBDAO.java, DBConnection.java**
  - LoginBean class act as our Model class and it is a normal java bean class, contains just setters and getters. Bean classes are efficiently used in java to access user information wherever required in the application and encapsulate many objects into a single object.
  - DBDAO class is the data access object to process sql statements to retrieve data from database.
  - DBConnection class makes a connection with database and contains JDBC code to connect with the MySQL database to retrieve data.

- **Web.xml**
  The web.xml is known as **a deployment descriptor**. It lists all the servlets used in the application. It is used to **mapping the servlet**.

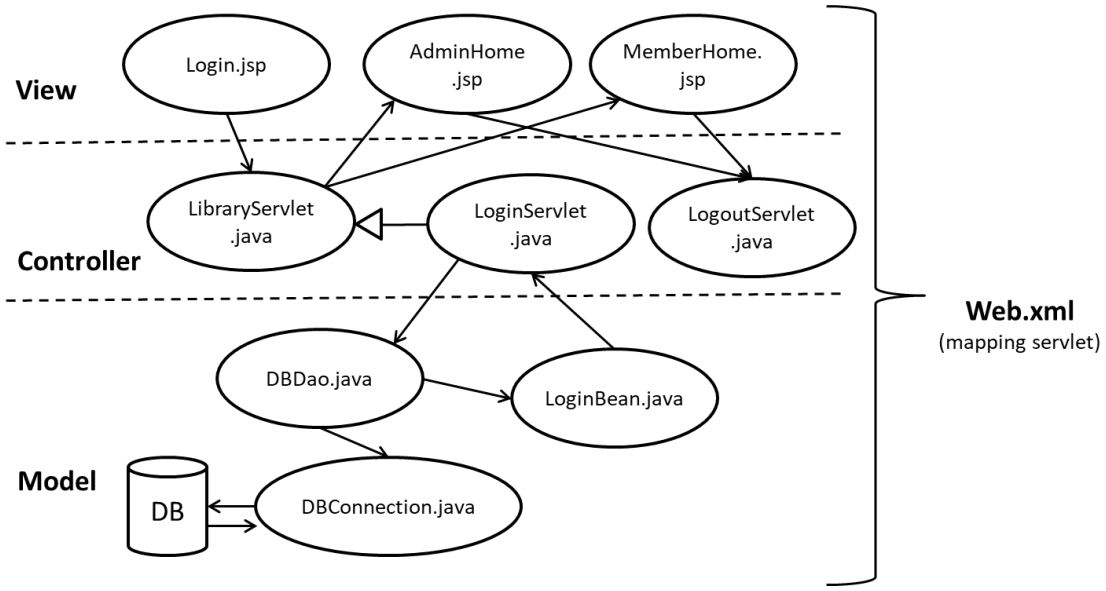


Figure 7.26: The Login Process using MVC pattern

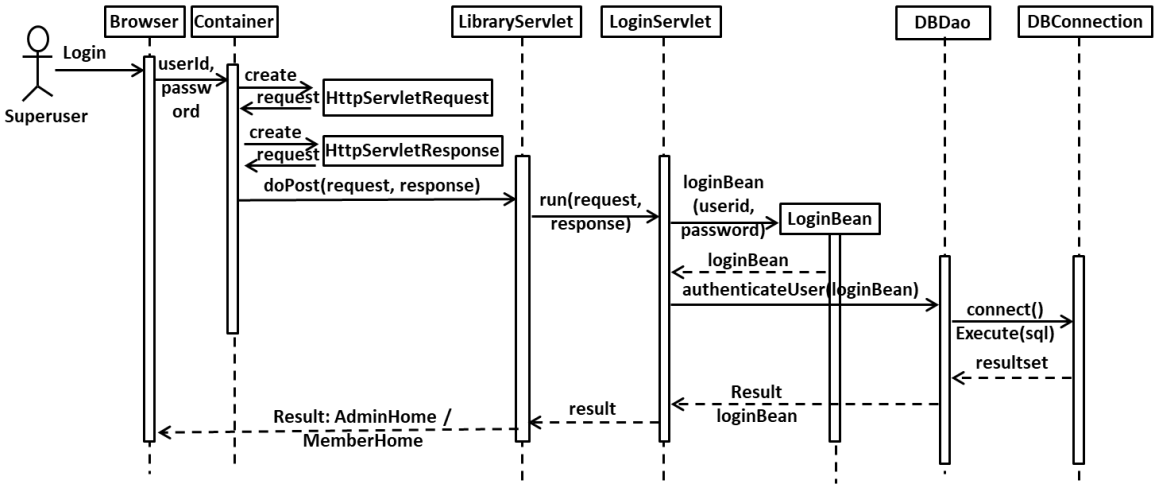
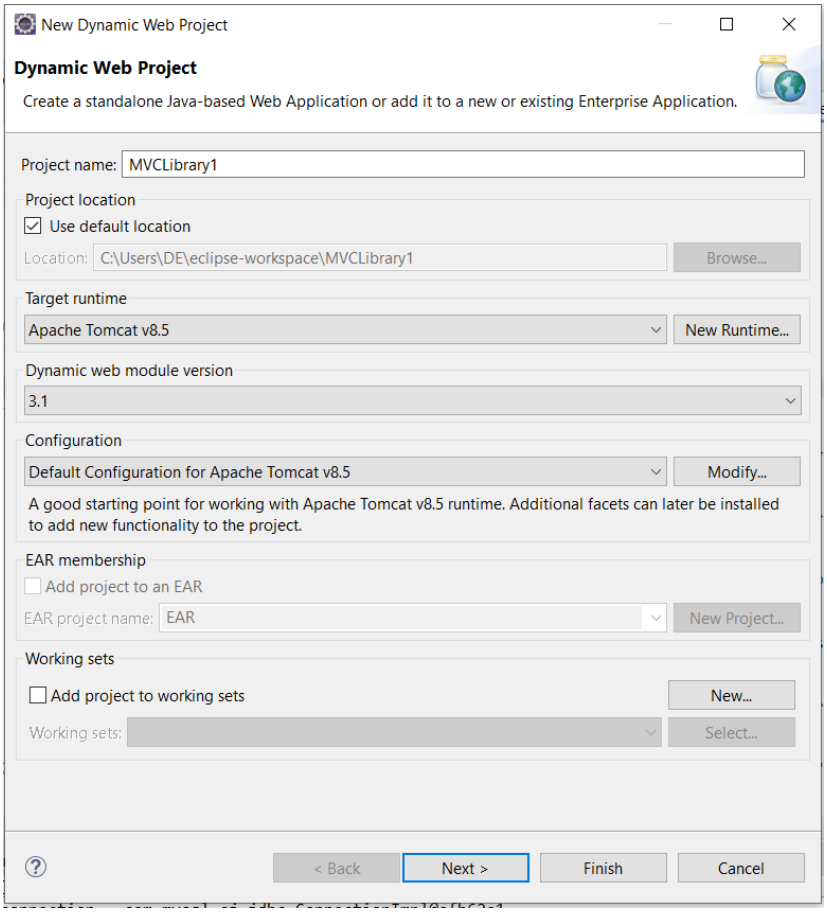
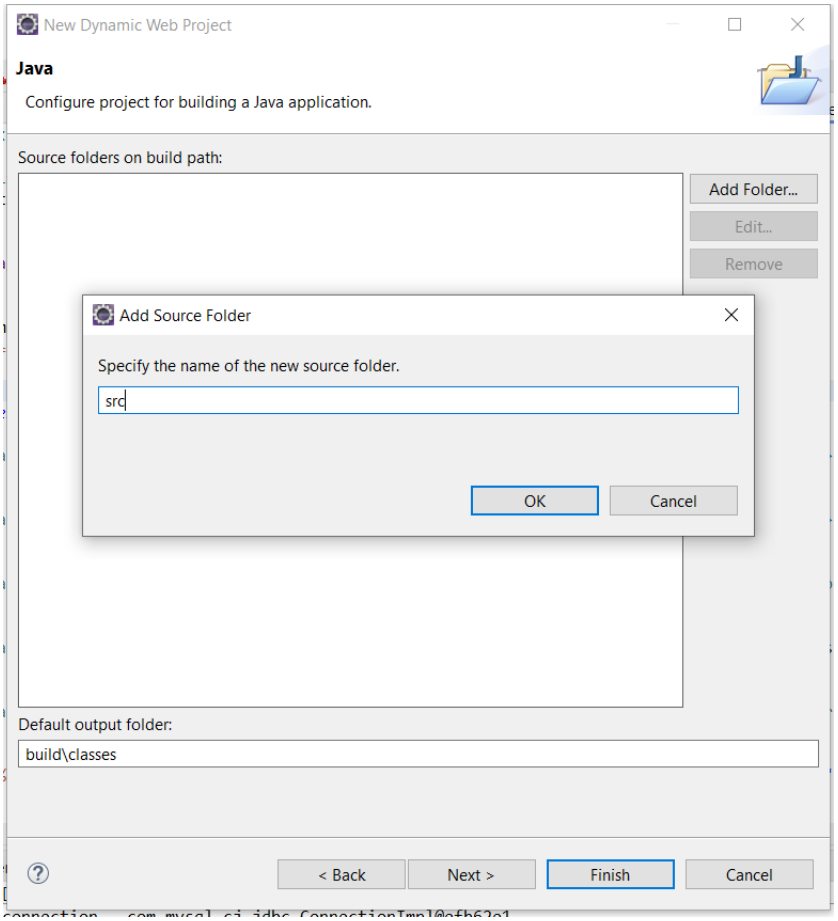
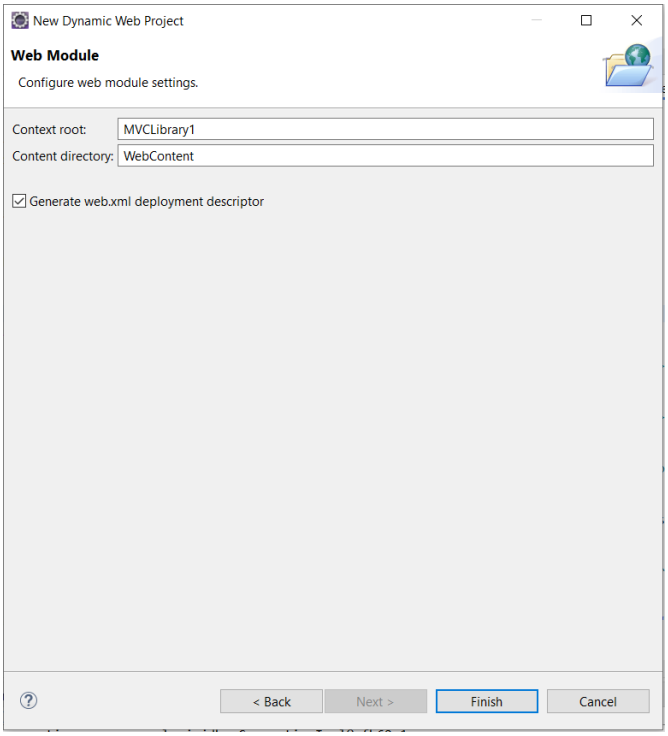


Figure 7.27: Sequence diagram for Role-based Login

Open eclipse, select New and create Dynamic Web Project. Give Project name: “MVCLibrary” and as shown below: create a new dynamic web project.


New Dynamic Web Project
Dynamic Web Project
Create a standalone Java-based Web Application or add it to a new or existing Enterprise Application.
Project name: MVCLibrary1
Project location
Use default location
Location: C:\Users\DE\eclipse-workspace\MVCLibrary1
Browse...
Target runtime
Apache Tomcat v8.5
New Runtime...
Dynamic web module version
3.1
Configuration
Default Configuration for Apache Tomcat v8.5
Modify...
A good starting point for working with Apache Tomcat v8.5 runtime. Additional facets can later be installed to add new functionality to the project.
EAR membership
Add project to an EAR
EAR project name: EAR
New Project...
Working sets
Add project to working sets
New...
Working sets:
Select...
< Back
Next >
Finish
Cancel


New Dynamic Web Project
Java
Configure project for building a Java application.
Source folders on build path:
Add Folder...
Edit...
Remove
Add Source Folder
Specify the name of the new source folder.
src
OK
Cancel
Default output folder:
build\classes
< Back
Next >
Finish
Cancel



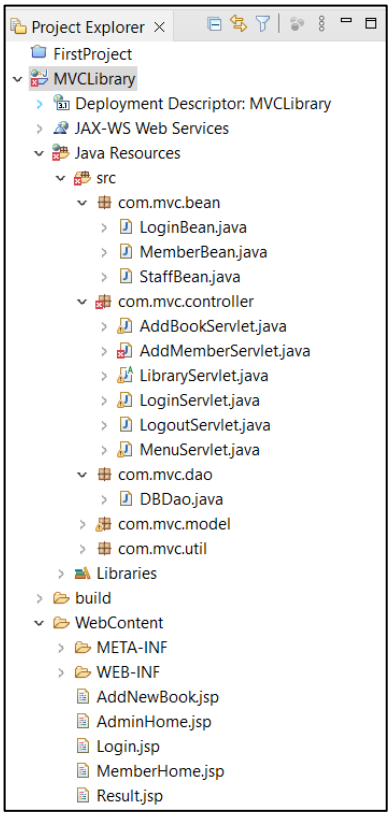


Figure 7.28: Structure of the project

Create new jsp file for "Login.jsp" as follow:

The Login.jsp is the view file that accepts user id and password. Here we use simple table to display the input. We could use CSS to format the page for more look and feels. The control servlet for this page is "LoginServlet.java".

The following statement is used to link between the jsp and it control servlet file.

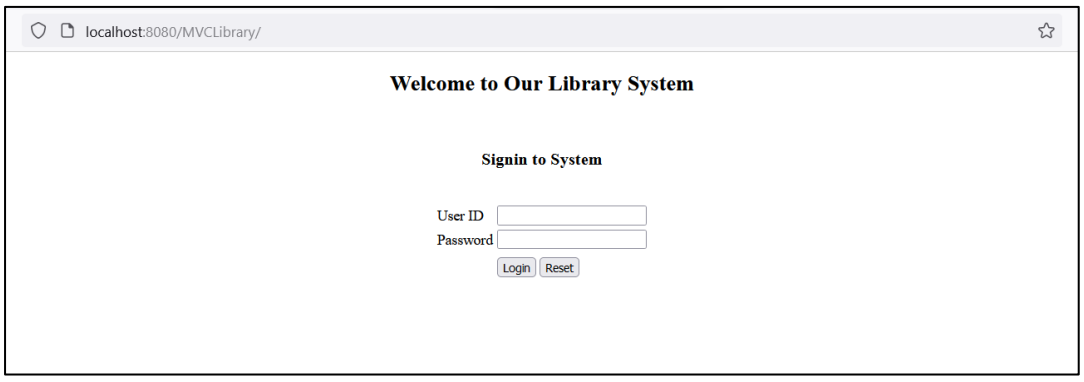
```
<form name="form" action="<%=request.getContextPath()%>/LoginServlet"
method="post">
```

The following jsp statement is used to output the error message from servlet to the jsp view.

```
<%=(request.getAttribute("errMessage") == null) ? "" :
request.getAttribute("errMessage")%>
```

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>MVCLibrary Login</title>
</head>
<body>
      <center><h2>Welcome to Our Library System</h2><br>
            <h3>Signin to System</h3><br>
      </center>
      <form name="form" action="<%=request.getContextPath()%>/LoginServlet"
method="post">
               <table align="center">
               <tr>
               <td>User ID</td>
               <td><input type="text" name="userid" /></td>
               </tr>
               <tr>
               <td>Password</td>
               <td><input type="password" name="password" /></td>
               </tr>
               <tr>
               <td><span style="color:red">
                    <%=(request.getAttribute("errMessage") == null) ? "" :
request.getAttribute("errMessage")%></span></td>
               </tr>
               <tr>
               <td></td>
               <td>
                    <input type="submit" value="Login"></input>
                    <input type="reset"  value="Reset"></input>
               </td>
               </tr>
               </table>
        </form>
    </body>
</html>
```

Output of Role-based Login



Create new jsp file for "AdminHome.jsp" as follow:

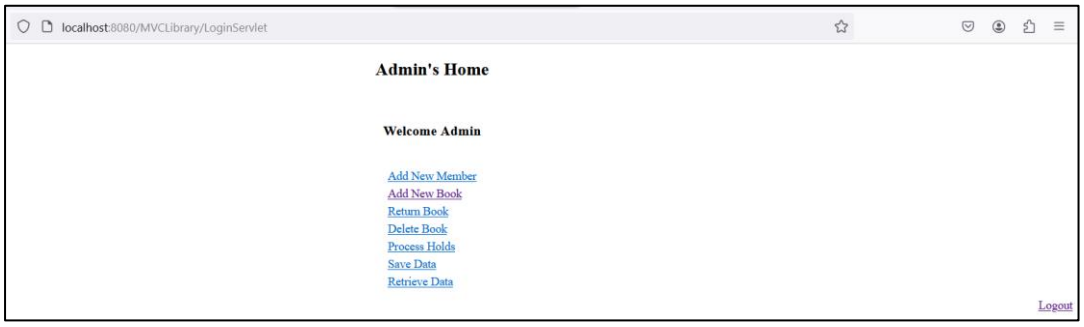
The AdminHome.jsp is the initial menu for super user after successful login. The each menu link sends the parameter value from URL link and control servlet is MenuServlet.java as follows:

```
<a href="<%=request.getContextPath()%>/MenuServlet?source=AddNewMember">Add New
Member</a>
```

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>SuperUser Home</title>
</head>
<% //In case, if Admin session is not set, redirect to Login page
      if((request.getSession(false).getAttribute("Admin")== null) )
      {
      %>
      <jsp:forward page="Login.jsp"></jsp:forward>
      <%} %>
      <body>
      <center>
            <h2>Admin's Home</h2><br>
            <h3>Welcome  <%=request.getSession(false).getAttribute("userName")
%> </h3><br>
      </center>

    <table align="center">
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=AddNewMember">Add New
Member</a> <br> </td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=AddNewBook">Add New
Book</a> <br></td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=ReturnBook">Return
Book</a> <br></td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=DeleteBook">Delete
Book</a><br></td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=ProcessHolds">Process
Holds</a><br></td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=SaveData">Save
Data</a><br></td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=RetrieveData">Retrieve
Data</a><br></td>
         </tr>
         <tr>
               <td><%=(request.getAttribute("errMessage") == null) ? "" :
request.getAttribute("errMessage")%></td>
         </tr>
         <tr>
         <td></td>
      </table>
```

```
      <div style="text-align: right"><a
href="<%=request.getContextPath()%>/LogoutServlet">Logout</a></div>
      </body>
</html>
```

Output of AdminHome.jsp (Admin Menu) after successful login of SuperUser



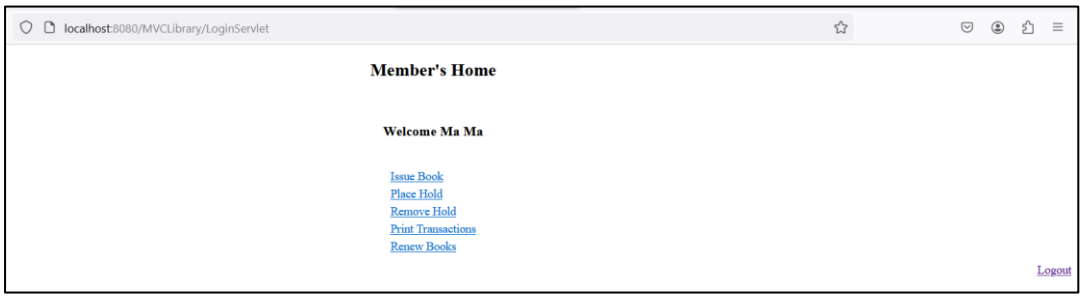
Create new jsp file for "MemberHome.jsp" as follow:

The MemberHome.jsp is the initial menu for the ordinary member.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Member Home</title>
</head>
      <% //In case, if Admin session is not set, redirect to Login page
      if((request.getSession(false).getAttribute("User")== null) )
      {
      %>
      <jsp:forward page="Login.jsp"></jsp:forward>
      <%} %>
      <body>
      <center><h2>Member's Home</h2><br />
      <h3>Welcome <%=request.getSession(false).getAttribute("userName") %>
</h3> <br />
      </center>

   <table align="center">
         <tr>
           <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=IssueBook">Issue
Book</a> <br></td>
         </tr>
         <tr>
            <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=PlaceHold">Place
Hold</a> <br></td>
         </tr>
         <tr>
           <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=RemoveHold">Remove
Hold</a> <br></td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=PrintTransactions">Prin
t Transactions</a> <br></td>
         </tr>
         <tr>
```

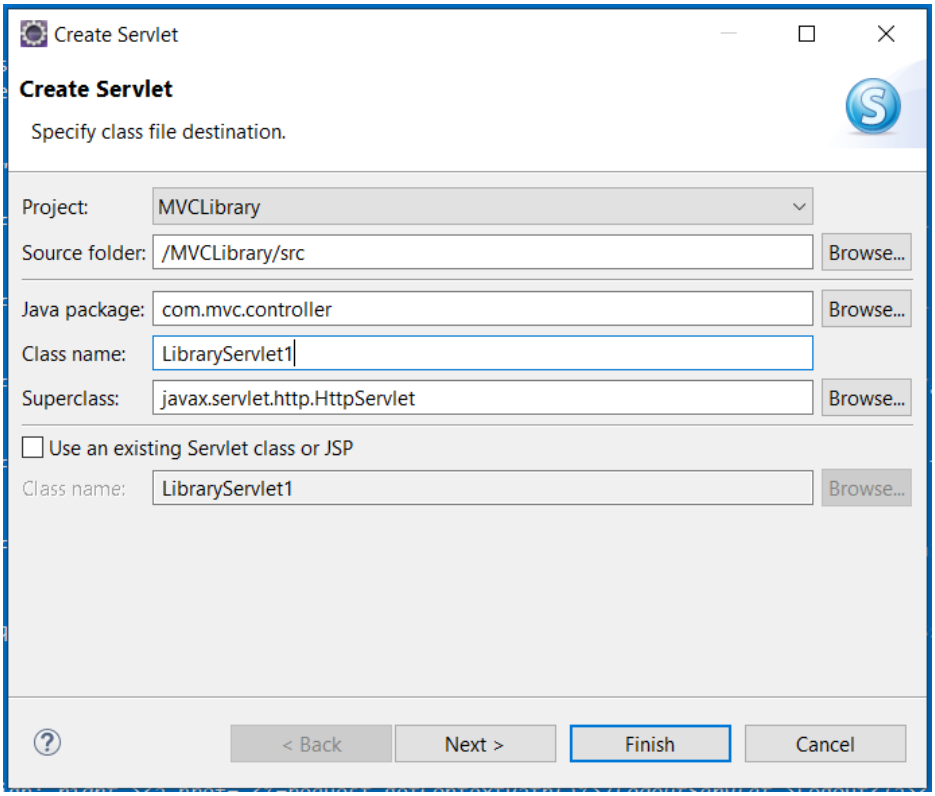
```
            <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=RenewBook">Renew
Books</a> <br></td>
          </tr>
          <tr>
            <td><%=(request.getAttribute("errMessage") == null) ? "" :
request.getAttribute("errMessage")%></td>
          </tr>
          <tr>
          <td></td>
       </table>
       <div style="text-align: right"><a
href="<%=request.getContextPath()%>/LogoutServlet">Logout</a></div>
       </body>
</html>
```

Output of MemberHome.jsp (Member Menu) after successful login of Ordinary Member



Create new servlet file : Controller – LibraryServlet.java

The LibraryServlet.java is only one servlet for the library system. All other servlets inherit from it and implement the run() abstract method to do their business process.



The LibraryServlet.java is only one servlet that is executed every time, user submit from the browser.

```
package com.mvc.controller;

import java.io.IOException;
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 javax.servlet.http.HttpSession;

/**
 * Servlet implementation class LibraryServlet
 */
@WebServlet("/LibraryServlet")
public abstract class LibraryServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    public LibraryServlet() {
        super();
    }

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

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

            String resultpage = run(request, response);
            System.out.println("Process output" + resultpage);
      }

      public boolean notLoggedIn(HttpServletRequest request) {
//checks user who may have invoked a command without actually login.
//return true if and only if the user has not currently logged in.
            return request.getSession(false) == null;
      }

      public String noLoginErrorMessage() {
// This method simply generates an HTML page that displays
//'Not logged in' and supplies a link to the log in screen.
// when a person who has not logged in attempts to execute a command.
            return "NoLogin.jsp";
      }
    /**
     * run() abstract method implemented by inherit class
     */
      public abstract String run(HttpServletRequest request,
HttpServletResponse response);

}
```

Create new servlet file: Controller – LoginServlet.java

The LoginServlet.java implements the **`run()` abstract method** of LibraryServlet.java, implementating the login process. It accepts the user inputs: userid and password from the Login.jsp and create the loginBean object. The loginBean object is passed to the DBDao.java to connect to the database and validate the user whether super user or ordinary member. After validation, if user is super user, the admin initial menu, AdminHome.jsp is displayed otherwise the member initial menu, MemberHome.jsp is diaplayed.

The statement "@WebServlet("/LoginServlet")" is used to map between view class "Login.jsp" and control servlet class "LoginServlet.java". If we put this statement in the servlet, we don't need to add map statement in "web.xml" file.

```
package com.mvc.controller;

import java.io.IOException;
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 javax.servlet.http.HttpSession;

import com.mvc.bean.LoginBean;
import com.mvc.dao.LoginDao;

/**
 * Servlet implementation class LoginServlet
 */

@WebServlet("/LoginServlet")

public class LoginServlet extends LibraryServlet {
      private static final long serialVersionUID = 1L;

    public LoginServlet() {
        super();
    }

       /**
        * Abstract method run() implementation
        */
    @Override
      public String run(HttpServletRequest request, HttpServletResponse
response) {

       String page="";
       String userid = request.getParameter("userid");
       String password = request.getParameter("password");

       //checks whether the user is a superuser or member
       LoginBean loginBean = new LoginBean();
       loginBean.setUserId(userid);
       loginBean.setPassword(password);

       LoginDao dbDao = new LoginDao();

       loginBean = dbDao.authenticateUser(loginBean);

       try {
             if(loginBean.getRole().equals("Admin")) {
       //Creating a session and setting session attribute
                HttpSession session = request.getSession();
                session.setAttribute("Admin", loginBean);
                session.setAttribute("userName",
                                        loginBean.getUserName());
             page = "AdminHome.jsp";
         request.getRequestDispatcher("AdminHome.jsp").forward(request,
response);
       }
             else if(loginBean.getRole().equals("Member")) {
           //Creating a session and setting session attribute
           HttpSession session = request.getSession();
           session.setMaxInactiveInterval(10*60);
           session.setAttribute("User", loginBean);
           session.setAttribute("userName", loginBean.getUserName());
```

```
            page = "MemberHome.jsp";
         request.getRequestDispatcher("MemberHome.jsp").forward(request,
response);
        }
        else {
        System.out.println("Error message = "+ loginBean.getRole());
        HttpSession session = request.getSession();
        session.setAttribute("errMessage", loginBean.getRole());
       page = "Login.jsp";
       request.getRequestDispatcher("Login.jsp").forward(request, response);
       }
   } catch (Exception e2) {
            e2.printStackTrace();
   }
   return page;
}

}
```

Create Model class file: LoginBean.java

The LoginBean.java is a normal java class, it acts as an data carrier object among controller, model, dao and database utility classes. It only have login attributes; userid, username, password and user role and setter and getter methods for these attributes.

```
package com.mvc.bean;

public class LoginBean {

    private String userid;
    private String username;
    private String password;
    private String role;

    public String getUserId() {
          return userid;
    }
    public void setUserId(String userId) {
          this.userid = userId;
    }
    public String getUserName() {
          return username;
    }
    public void setUserName(String userName) {
          this.username = userName;
    }

    public String getPassword() {
          return password;
    }
    public void setPassword(String password) {
          this.password = password;
    }
    public String getRole() {
          return role;
    }
    public void setRole(String role) {
          this.role = role;
    }
}
```

Create DAO class file: LoginDao.java

The LoginDao.java is a data access object between application and databases. It connects the DBConnection.java to get connect to database and get the resultset of the sql statements for Login process. The login process first check in the superuser table, if found then the superuser information is passed into the LoginBean object and return it. If not found, the check in the member table. If found, the member information is passed into the LoginBean object and return it. If both check is not found, the error message is returned.

```
package com.mvc.dao;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.mvc.bean.LoginBean;
import com.mvc.util.DBConnection;


public class LoginDao {

	public LoginBean authenticateUser(LoginBean loginBean)
	{
	    String userId = loginBean.getUserId();
	    String password = loginBean.getPassword();

	    Connection con = null;
	    Statement statement = null;
	    ResultSet resultSet = null;

	    String userIdDB = "";
	    String passwordDB = "";
	    boolean found = false;

	    try
	    {
	        con = DBConnection.createConnection();
	        statement = con.createStatement();
	        resultSet = statement.executeQuery("select staffid,name,password
from staffs");

	     // search in staffs
	    while(resultSet.next())
	    {
	       userIdDB = resultSet.getString("staffid");
	      passwordDB = resultSet.getString("password");

	      if(userId.equals(userIdDB) && password.equals(passwordDB))
	      {
	        loginBean.setUserName(resultSet.getString("name"));
	       loginBean.setRole("Admin");
	       found = true;
	       break;
	      }
	    }

	     if (!found) {  //search in members
	        resultSet.close();
	        resultSet = statement.executeQuery("select memberid,name,password
from members");
	        found = false;

	        while(resultSet.next())
	        {
	          userIdDB = resultSet.getString("memberid");
	          passwordDB = resultSet.getString("password");
```

```
                if(userId.equals(userIdDB) && password.equals(passwordDB)) {

                  loginBean.setUserName(resultSet.getString("name"));
                 loginBean.setRole("Member");
                 found = true;
                  break;
                }
               }
        }
        if(!found)
               loginBean.setRole("Invalid user credentials");
    }
    catch(SQLException e)
    {
         e.printStackTrace();
    }
     return loginBean;
  }

}
```

Create Utility class file: DBConnection.java

The DBConnection.java is the connection class for the database. It opens connection to the MySql database, called “mylibrarydb” with username = “root” and password=””.

```
package com.mvc.util;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBConnection {

  public static Connection createConnection(){

    Connection con = null;
    String url = "jdbc:mysql://localhost:3306/mylibrarydb";
    String username = "root";
    String password = "";
    try
    {
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
        }
        catch (ClassNotFoundException e)    {
            e.printStackTrace();
        }
        con = DriverManager.getConnection(url, username, password);
        System.out.println("Post establishing a DB connection-"+con);
    }
    catch (SQLException e)    {
        System.out.println("An error occurred. Maybe user/password is
invalid");
        e.printStackTrace();
    }
    return con;
  }
}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
```

```
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" id="WebApp_ID"
version="3.1">
  <display-name>MVCLibrary</display-name>
  <welcome-file-list>
    <welcome-file>Login.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

5.3.3 Add New Book

The **Add New Book page** is **super-user access right to insert new book information,** after super-user has successfully login in. The super-user's initial menu display shown in Figure 7.xx(b) and super user choose "Add New Book" menu. The adding new book page is displayed to accept the new book information.

In our previous standalone library system, the sequence diagram for adding new books is shown in Figure 7.29 (a). Now in the web based system, the sequence diagram for adding new books is shown in Figure 7.29(b). Notice that the interface class is replaced with the interaction of view and web container and instead of adding into the Catalog class, the new book is saved into the database. All new book data are added into the database through BookDao class instead of the Catalog class. BookDao class creates the sql statements to save data into database. DBConnection class makes a connection with the MySQL database to save data. Here, Library class also acts as the single entry point to access all other classes.

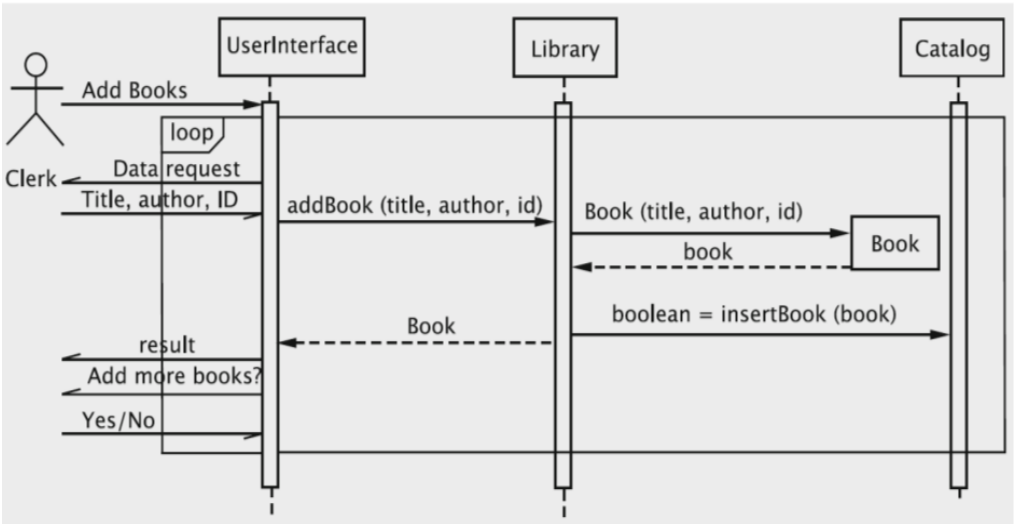


Figure 7.29(a): Sequence diagram for adding books in standalone system

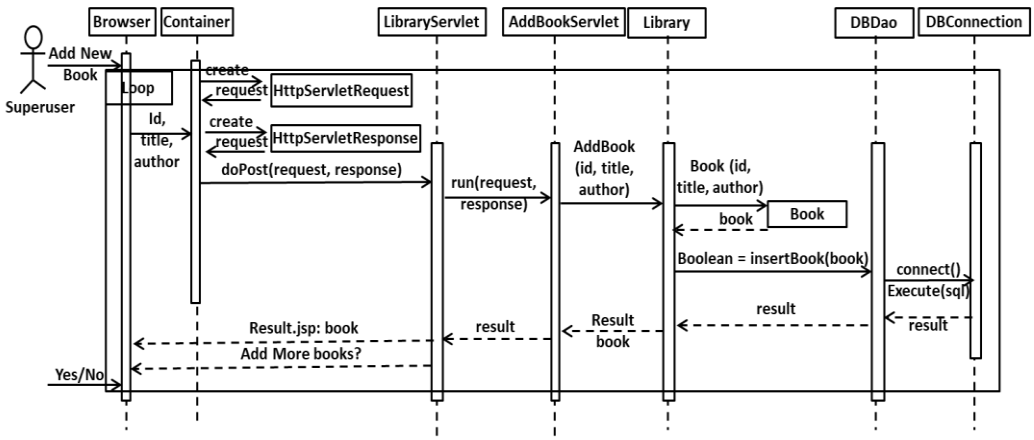


Figure 7.29(b): Sequence diagram for adding books in web-based system

According to the MVC pattern, we will create the following files for add new book process;

- **View - AddNewBook.jsp, Result.jsp**
  - The AddNewBook.jsp contains a simple HTML form to key in new book information.
  - The Result.jsp contains successful insertion and asking more book to insert.

- **Controller – LibraryServlet.java, MenuServlet.java, AddBookServlet.java**
  - LibraryServlet.java is only one servlet file which acts as container to accept multiple requests from browser and menu selection details are forwarded to corresponding servlet class, for adding new book call AddBookServlet.java.

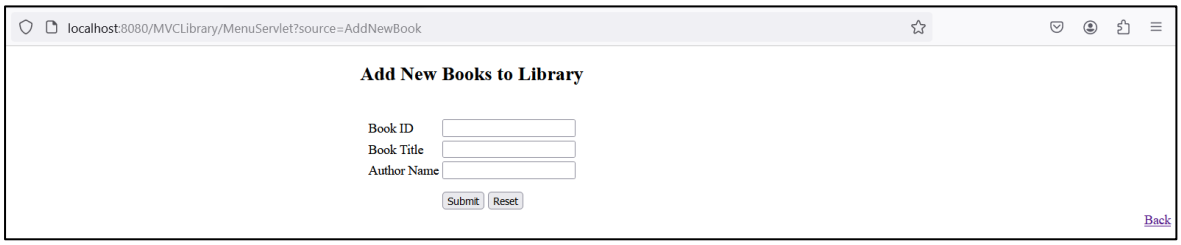
- **Model – Library.java, Book.java, BookDao.java and DBConnection**
  - Library.java class act as control to access all model classes.
  - Book.java contains setters and getters, borrwedby and duedate.
  - BookDAO class creates the sql statements to save data into database.
  - DBConnection class makes a connection to MySQL database using JDBC code to save data.

Create **AddNewBook.jsp** as follows. **Java script** is used to check each input is null or not.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
      <title>Add New Book</title>
      <script>
      function validate()
      {
            var bookid = document.form.bookid.value;
            var title = document.form.title.value;
            var author = document.form.author.value;

            if (bookid==null || bookid=="") {
             alert("Book ID can't be blank");
             return false;
            }
            else if (title==null || title==""){
             alert("Book Title can't be blank");
             return false;
            }
            else if (author==null || author==""){
             alert("Book Author can't be blank");
             return false;
            }
      }
      </script>
</head>
<body>
      <center><h2>Add New Books to Library</h2><br /></center>
          <form name="form"
action="<%=request.getContextPath()%>/AddBookServlet?source=insert"
method="post" onsubmit="return validate()">
              <table align="center">
               <tr>
               <td>Book ID</td>
               <td><input type="text" name="bookid" /></td>
               </tr>
               <tr>
               <td>Book Title</td>
               <td><input type="text" name="title" /></td>
               </tr>
```

```
                <tr>
                <td>Author Name</td>
                <td><input type="text" name="author" /></td>
                </tr>
                <tr></tr>
                <tr>
                <td><%=(request.getAttribute("Message") == null) ? "" :
request.getAttribute("Message")%></td>
                </tr>
                <tr>
                <td><%=(request.getAttribute("errMessage") == null) ? "" :
request.getAttribute("errMessage")%></td>
                </tr>
                <tr>
                <td></td>
                <td>
                    <input type="submit" value="Submit"></input>
                    <input type="reset"  value="Reset"></input>
                </td>
                </tr>
               </table>
           </form>
           <div style="text-align: right"><a
href="<%=request.getContextPath()%>/AddBookServlet?source=AdminHome">Back</a></d
iv>
       </body>
</html>
```

Output of the AddNewBook.jsp

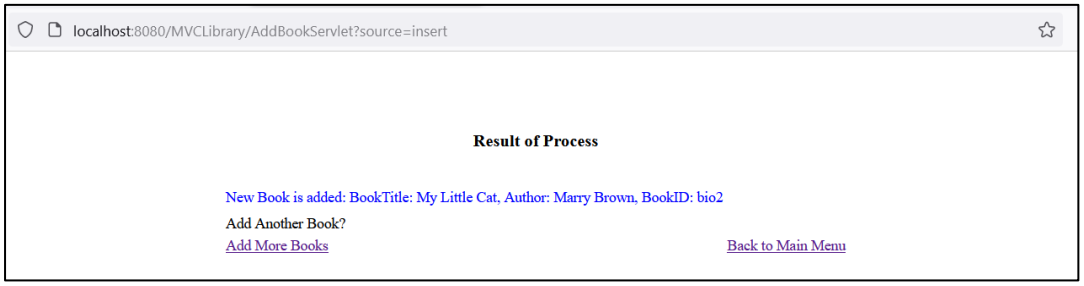


Create **Result.jsp** to output the successful insertion or not and any more new book to input.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Result</title>
</head>
<body>
      <br /><br /><br />
      <center><h3>Result of Process</h3><br /></center>
      <table align="center">
          <tr>
               <td><span style="color:blue">
                   <%=request.getAttribute("Message") %></span><br />
               </td>
          </tr>
          <tr> </tr> <tr> </tr>
          <tr>
               <td>Add Another Book? </td>
          </tr>
          <tr>
               <td>
                   <a
href="<%=request.getContextPath()%>/AddNewBook.jsp">Add More Books</a>
```

```
                </td>
                <td>
                    <a
href="<%=request.getContextPath()%>/AdminHome.jsp">Back to Main Menu</a>
                </td>
            </tr>
            <tr></tr>
            <tr></tr>
        </table>
</body>
</html>
```

Output of the Result.jsp



Update Controller LibraryServlet.java file as follows:

Add the **Library** variable and initialization in the **LibraryServlet.java.** The rest of the code is same as before.

```
    protected Library library;

    public void init() {
          if (library== null)
                library = new Library();
    }
```

```
package com.mvc.controller;

import java.io.IOException;
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 javax.servlet.http.HttpSession;

/**
 * Servlet implementation class LibraryServlet
 */
@WebServlet("/LibraryServlet")
public abstract class LibraryServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;
    protected Library library;

    public void init() {
       if (library== null)
             library = new Library();
    }

    public LibraryServlet() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse
```

```
response) throws ServletException, IOException {

		doPost(request, response);
	}

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

		String result = run(request, response);
		System.out.println("Process output" + result);
	}

	public boolean notLoggedIn(HttpServletRequest request) {
//checks user who may have invoked a command without actually login.
//return true if and only if the user has not currently logged in.
		return request.getSession(false) == null;
	}

	public String noLoginErrorMessage() {
// This method simply generates an HTML page that displays
//'Not logged in' and supplies a link to the log in screen.
// when a person who has not logged in attempts to execute a command.
		return "NoLogin.jsp";
	}

    /**
     * run() abstract method implemented by inherit class
     */
	public abstract String run(HttpServletRequest request,
HttpServletResponse response);

}
```

Create Controller – MenuServlet.java

The **MenuServlet.java** is the controller servlet **class for the Superuser's initial menu** and it get the parameter and forwards to user selected jsp file to display.

```
package com.mvc.controller;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * Servlet implementation class MenuServlet
 */
@WebServlet("/MenuServlet")
public class MenuServlet extends LibraryServlet {
	private static final long serialVersionUID = 1L;

	/**
	 * Abstract run() method implementation
	 */
	public String run(HttpServletRequest request, HttpServletResponse
response) {

	String source = request.getParameter("source") + ".jsp";

	try {
	request.getRequestDispatcher(source).forward(request, response);
	} catch (Exception e2)    {
			e2.printStackTrace();
	}
	  return source;
```

```
        }
}
```

Create Controller – AddBookServlet.java

The AddBookServlet.java is the business logic class it implement the run() abstract method and send user input data to the Library class to add new book information and return the successful/fail result of adding book to display in Result.jsp file.

```
package com.mvc.controller;

import java.io.IOException;
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 com.mvc.model.Library;
import com.mvc.model.Book;

/**
 * Servlet implementation class AddBookServlet
 */
@WebServlet("/AddBookServlet")
public class AddBookServlet extends LibraryServlet {
    private static final long serialVersionUID = 1L;

    public AddBookServlet() {
        super();
    }

    public String run(HttpServletRequest request, HttpServletResponse
response) {

       String source = request.getParameter("source");
       try
       {
          if (source.equals("insert"))
          {
              Book result;
              String title = request.getParameter("title");
              String author = request.getParameter("author");
              String bookID = request.getParameter("bookid");

              result = library.addBook(title, author, bookID);

              if (result != null) {
                 request.setAttribute("Message", "New Book is added: "  +
result.toString());
              } else {
                 request.setAttribute("Message", " Oops.. Something went wrong
in adding new book..!!!");
              }

request.getRequestDispatcher("Result.jsp").forward(request, response);
              source = "Result.jsp";
           }
           else if (source.equals("AdminHome"))
          {
         request.getRequestDispatcher("AdminHome.jsp").forward(request,
response);
              source = "AdminHome.jsp";
       }
       } catch (Exception e2)    {
              e2.printStackTrace();
       }
```

```
        return source;
    }
}
```

Reuse Model class – Library.java, Book.java, BookDao.java,

This Library.java, Book.java are reused classes that we have created in the stand alone project in lecture 3.

Library.java

```
package com.mvc.model;

import java.util.*;
import java.io.*;
import com.mvc.dao.BookDao;

public class Library {

    // The Adding New Books process
    public Book addBook (String title, String author, String bID) {

       Book  book = new Book(title, author, bID);

       //add to database
       BookDao  bookDao = new BookDao();
       String result = bookDao.insertBook(book);
       if (result.equals("SUCCESS"))
             return (book);
       return null;
    }
    //some code are not shown
    …
 }
```

Book.java

```
package com.mvc.model;

import java.util.*;
import java.io.*;

public class Book {

       private  String  title, author, bookId;

       public Book (String title, String author, String bId) {
              this.title = title;
              this.author = author;
              this.bookId = bId;
       }

       public  String   getTitle() {  return this.title;   }
       public  String   getAuthor() { return this. author; }
       public  String   getBookId() { return this.bookId; }

       public  void   setTitle(String btitle) {
              this.title = btitle;
       }
       public  void   setAuthor(String bauthor) {
              this. author = bauthor;
       }
       public  void   setBookId(String bId) {
               this.bookId = bId;
       }

       @Override
```

```
	public  String toString() {
		return "BookTitle: " + this.title  + ", Author: " +
			this.author  + ", BookID: " +  this.bookId;
	}

	//some code are not shown
}
```

BookDao.java

```
package com.mvc.dao;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.Statement;

import com.mvc.util.DBConnection;
import com.mvc.model.Book;

public class BookDao {

	public String insertBook(Book book)
	{
		Connection con = null;
		PreparedStatement statement = null;

	    try
	    {
		    con = DBConnection.createConnection();

		  //Insert book details into the table 'book'
		    String query = "insert into book(bookid,title,author) values
(?,?,?)";

		  //Making use of prepared statements here to insert bunch of data
		    statement = con.prepareStatement(query);
		    statement.setString(1, book.getBookId());
		    statement.setString(2, book.getTitle());
		    statement.setString(3, book.getAuthor());

		    int i= statement.executeUpdate();
		  //Just to ensure data has been inserted into the database
		    if (i!=0)
		    {
			    return "SUCCESS";
		    }
	    }
	    catch(SQLException e)
	    {
			e.printStackTrace();
	    }
	   // On failure, send a message from here.
	    return "Oops.. Something went wrong there..!";
	}
}
```

Similarly, adding new member is same process as adding new book. The rest of the process implementations are left as exercise or assignment for student.

5.4 Remembering Users

Servlets typically deal with multiple users. When a servlet receives data from a browser, it must somehow figure out which user sent the message, what the user's privileges are, etc. **Each request from the browser to the server starts a new connection**, and **once the request is served, the connection is torn down**.

However, typical **web transactions involve multiple request–response pairs**. This makes **the process of remembering the user associated with a connection** somewhat difficult without extra support from the system. The system provides the necessary support by means of what are known as **sessions**, which are of type **HttpSession**. **When it receives a request from a browser**, the servlet may call the method **`getSession()`** on the **HttpServletRequest** object to **create a session object**, or if a session is already associated with the request, to get a reference to it. **To check if a session is associated with the request** and to optionally create one, a variant of this method **`getSession(boolean create)`** may be used. If the **value false** is passed to this method and the **request has no valid HttpSession**, this **method returns null**. When a user logs in, the system creates a session object as below.

```
HttpSession session = request.getSession();
```

When the user logs out, the session is removed as below.

```
session.invalidate();
```

Requests other than log in require the user to be logged in. The following code evaluates to **true if** the user does not have a session: that is, the **user has not logged in**.

```
request.getSession(false) == null
```

A **session object** can be used to **store information about the session**. In our library system, we would like to **store** the **user id, the type of terminal from which the user has logged in,** and **some additional information** related to the user. The methods for this are

1. **`void setAttribute(String name, Object value)`**

   This command binds value, the object given in the second parameter, to the attribute specified in name. By setting the **second parameter to null**, the **attribute can be removed**.

2. **`Object getAttribute(String name)`**

   The attribute value associated with name is returned.

3. **`void removeAttribute(String name)`**

   This method deletes the specified attribute from this session.

For handling users who may have invoked a command without actually logging in, the method `notLoggedIn()` returns **true** if and only if the user has not currently logged in. The check for whether the user has logged in as follows:

```
public boolean notLoggedIn(HttpServletRequest request) {
    return request.getSession(false) == null;
}
```

The method `noLoginErrorMessage()` returns HTML code that displays an error message **when a person who has not logged in attempts to execute a command**. This method simply generates an HTML page that displays 'Not logged in' and redirect a link to the log in screen.

```
public String noLoginErrorMessage() {
      return "NoLogin.jsp";
}
```

5.4.1 Logging In and Logging Out

In our library system, the web app is **run on local area network of the Library**. The **IP addresses of all client machines located in the library** are listed in **a table named "IPAddresses".** The list of super-users and their passwords is stored in a file named "staffs" table in database.

When the user login into the system, control goes to the `Loginservlet` that inherit from a `LibraryServlet` which receives data from a browser through a **HttpServletRequest object**. This involves **parameter names** and **their values**, **IP address of the user**, and **so on**.

We now discuss **how we remember the member and super-user who had logged into the library**. If the superuser id is valid, it is remembered in the attribute `currentUser` which stores the `loginBean` object. We also remember the attribute `UserType`, when a super-user is logged in, the value of this attribute `UserType` is the string ("Superuser").
If the Superuser id is valid, it is remembered in the attribute `currentUser` as follow:

```
HttpSession session = request.getSession();
session.setAttribute("UserType", "Superuser");
session.setAttribute("currentUser", loginBean);
```

We retrieve the `currentUser` as follows:

```
LoginBean loginBean = session.getAttribute(request, "currentUser");
String superuserid = loginBean.getUserId();
String superusername = loginBean.getUserName();
String role = loginBean.getRole();
```

We have already discussed in the previous section of **Role-based Login**. In the code of `LoginServlet` class, the `run()` method checks if the user is a superuser by calling the method `LoginDao.authenticateUser(loginBean)` that validates Super-user and if so, the attribute `UserType` is given the value "`Superuser`", otherwise, validate if the user is a member of the library; in that case, the `UserType` attribute is set as "`Member`". **In such a way, the system remembers the login user using the session attributes.**

With a successful log-in, the method also **checks whether the terminal used is within the library premises or outside**. First, the code get the ipaddress of current client request and check in database ipaddresses table. Then the attribute `location` save whether user login from Library or Outside. See code snipp as follows:

```
//Check for access location
//first get the ip address of client request
String ipAddress = request.getRemoteAddr();

System.out.println("IPAddress: " + ipAddress);
if (loginDao.libraryInvocation(ipAddress)) {
   session.setAttribute("location", "Library");
} else {
   session.setAttribute("location", "Outside");
   session.setAttribute("errMessage", "Please Login from Library Terminal
!!");
}
   page = "AdminHome.jsp";
   request.getRequestDispatcher("AdminHome.jsp").forward(request, response);
```

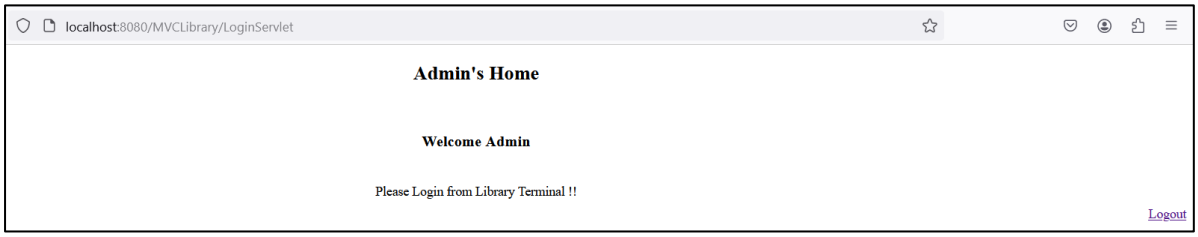
The update code for `LoginServlet` class is shown below:

```
package com.mvc.controller;

import java.io.IOException;
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 javax.servlet.http.HttpSession;

import com.mvc.bean.LoginBean;
import com.mvc.dao.DBDao;

/**
 * Servlet implementation class LoginServlet
 */
@WebServlet("/LoginServlet")
public class LoginServlet extends LibraryServlet {

    private static final long serialVersionUID = 1L;

    public LoginServlet() {
        super();
    }
       /**
        * Abstract run() method implementation
        */
    @Override
   public String run(HttpServletRequest request,
                                     HttpServletResponse response) {

     try
     {
       //check there is already logged in user,
       //if so redirect to the last page visited
      if (!notLoggedIn(request)) {
            response.sendRedirect(LastPage);
            return LastPage;
      }

      String userid = request.getParameter("userid");
      String password = request.getParameter("password");

      //checks whether the user is a superuser or member
      LoginBean loginBean = new LoginBean();
      loginBean.setUserId(userid);
      loginBean.setPassword(password);

      //check in the database
      LoginDao loginDao = new LoginDao();
      loginBean = loginDao.authenticateUser(loginBean);

      try {
         if(loginBean.getRole().equals("Admin")) {
      //Creating a session and setting session attribute
            HttpSession session = request.getSession();
            session.setAttribute("UserType", "Superuser");
            session.setAttribute("currentUser", loginBean);
           session.setAttribute("userName", loginBean.getUserName());
            session.setAttribute("errMessage", "");

      //Check for access location
            String ipAddress = request.getRemoteAddr();
if (loginDao.libraryInvocation(ipAddress)) {
              session.setAttribute("location", "Library");
```

```
            } else {
              session.setAttribute("location", "Outside");
              session.setAttribute("errMessage", "Please Login from Library
Terminal !!");
            }
            LastPage = "AdminHome.jsp";
        request.getRequestDispatcher("AdminHome.jsp").forward(request,
response);
            }
          else if(loginBean.getRole().equals("Member")) {
              HttpSession session = request.getSession();
              session.setMaxInactiveInterval(10*60);
              session.setAttribute("UserType", "Member");
              session.setAttribute("currentUser", loginBean);
              session.setAttribute("userName", loginBean.getUserName());
              session.setAttribute("errMessage", "");

      //Check for access location
            String ipAddress = request.getRemoteAddr();
            System.out.println("IPAddress: " + ipAddress);
            if (loginDao.libraryInvocation(ipAddress)) {
               session.setAttribute("location", "Library");
             } else {
               session.setAttribute("location", "Outside");
               session.setAttribute("errMessage", "To IssueBook, Please Login
from Library Terminal !!");
                  }
                LastPage = "MemberHome.jsp";
   request.getRequestDispatcher("MemberHome.jsp").forward(request, response);
          }  else {
            System.out.println("Error message = "+ loginBean.getRole());
            HttpSession session = request.getSession();
            session.setAttribute("errMessage", loginBean.getRole());
            LastPage = "Login.jsp";
      request.getRequestDispatcher("Login.jsp").forward(request, response);
          }
        } catch (Exception e2) {
           e2.printStackTrace();
        }
        return LastPage;
      }
}
```

The final step is to forward to the appropriate menu jsp page. These meet the requirements we set forth under 'Developing User Requirements'. If the user has logged in from the library, the `Issue Book` command is inserted into the menu. For super users, commands such as Add and Remove Book are inserted. Ordinary members always get to issue commands such as placing a hold and removing a hold. These commands are also available to superusers who log in from a library terminal. Finally, the logout command is available to all users from anywhere.

For superuser login, the AdminHome.jsp is displayed and it checks the `location` attribute whether `Library` or `Outside`. The `login location` is checked with the client's `ipaddress`. When the superuser login from the outside of the library, only the `logout` command is showed. The output is shown as below.



The update code for AdminHome.jsp is as follow:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>SuperUser Home</title>
</head>

<% //In case, if user session is not set, redirect to Login page
if((request.getSession(false).getAttribute("currentUser")== null) )
{ %>
            <jsp:forward page="Login.jsp"></jsp:forward>
<%} %>

<body>
<center>
      <h2>Admin's Home</h2><br>
      <h3>Welcome  <%=request.getSession(false).getAttribute("userName") %>
</h3><br>
</center>

    <table align="center">

<%  //In case, if Superuse login from outside of libray, only logout command
is available
    if((request.getSession(false).getAttribute("location")=="Library")) {  %>
        <tr>
         <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=AddNewMember">Add New
Member</a> <br> </td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=AddNewBook">Add New
Book</a> <br></td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=ReturnBook">Return
Book</a> <br></td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=DeleteBook">Delete
Book</a><br></td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=ProcessHolds">Process
Holds</a><br></td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=SaveData">Save
Data</a><br></td>
         </tr>
         <tr>
               <td><a
href="<%=request.getContextPath()%>/MenuServlet?source=RetrieveData">Retrieve
Data</a><br></td>
         </tr>
         <%} %>
         <tr>
               <td><%=(request.getSession(false).getAttribute("errMessage") ==
```

```
null) ? "" : request.getSession(false).getAttribute("errMessage")%></td>
          </tr>
          <tr>
          <td></td>
       </table>

       <div style="text-align: right"><a
href="<%=request.getContextPath()%>/LogoutServlet">Logout</a></div>
       </body>
</html>
```

When user logout from the library system, all session values are deleted, the last page variable is set to null and login page is redirected. The code in the `LogoutServlet` class is as follows:

```
package com.mvc.controller;

import java.io.IOException;
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 javax.servlet.RequestDispatcher;
import javax.servlet.http.HttpSession;

/**
 * Servlet implementation class LogoutServlet
 */
@WebServlet("/LogoutServlet")
public class LogoutServlet extends LibraryServlet {
       private static final long serialVersionUID = 1L;

    /**
        * @see HttpServlet#doGet(HttpServletRequest request,
HttpServletResponse response)
        */
       public String run(HttpServletRequest request, HttpServletResponse
response)  {

       try
       {
              HttpSession session = request.getSession(false);
             //Fetch session object

              if(session!=null) //If session is not null
              {
                //removes all session attributes bound to the session
                 session.invalidate();
                 LastPage = null;
                 request.setAttribute("errMessage", "You have logged out
successfully");
                 RequestDispatcher requestDispatcher =
request.getRequestDispatcher("Login.jsp");
                 requestDispatcher.forward(request, response);
                 System.out.println("Logged out");
              }
       } catch (Exception e2)    {
                 e2.printStackTrace();
       }
       return LastPage;
  }
}
```

5.4.2 Execution Flow to Check the Remembering User

To the best understood of remembering user, we need to understand the cases. There are possible cases for user who connects to the library from a different window of the browser. Some cases are as follows:

(1) When the user has **two windows connected to the library** and the **exit command is issued from one of the two**.
(2) When the user **tries to log in from a different window of the browser**.
(3) When the user overwrites the current page by visiting some other site and **wants to come back to the library system**.

Execution Flow Case 1:

To solve the case 1, we need to **check before every processing start that the user is login or not** because when the user has two windows connected to the library and the exit command is issued from one of the two and try to execute a command (insert) in one window after logged out from one window.

To understand the case (1), we explain with the previous process of adding new book as example. Since add new book command is only for the superuser, after the superuser is successful login to the system, the superuser menu "AdminHome.jsp" is displayed.

For example, when **add new book command is chosen from one window** and the **adding new book page is displayed at another browser** that asks the user to enter the **book id, title and author of a book** that should be added or click on a link to reset the command and return to the main menu, shown in Figure 7.xx. The possible case is happened, when user logouts from the one window and try to insert the new book in another window. Therefore we need to check whether user is login or not before inserting book into the database.

To handle this case, we check with the `notLoggedIn(request)` method in the `run()` method of `AddBookServlet` class. If user is already logout, the `NoLogin.jsp` page is displayed where the nologin error message is displayed and ask user to login again. The code snip is shown below:

```
if (notLoggedIn(request)) {
      return "NoLogin.jsp";
}
```

The updated code for the AddBookServlet.java is as follows:

```
package com.mvc.controller;

import java.io.IOException;
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 javax.xml.transform.Source;

import com.mvc.model.Library;
import com.mvc.model.Book;

/**
 * Servlet implementation class AddBookServlet
 */
@WebServlet("/AddBookServlet")
public class AddBookServlet extends LibraryServlet {
      private static final long serialVersionUID = 1L;

    public AddBookServlet() {
        super();
    }
```

```
    public String run(HttpServletRequest request, HttpServletResponse response)
{

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

       try
       {
              if (notLoggedIn(request)) {
                     return "NoLogin.jsp";
              }

              if (source.equals("insert"))
              {
                     Book result;

                     String title = request.getParameter("title");
                     String author = request.getParameter("author");
                     String bookID = request.getParameter("bookid");

                     result = library.addBook(title, author, bookID);

                     if (result != null) {
                       request.setAttribute("Message", "New Book is added: "  +
result.toString());
                     } else {
                        request.setAttribute("Message", "Oops.. Something went
wrong in adding new book..!!!");
                     }
                     source = "Result.jsp";
              }
              else if (source.equals("AdminHome"))
              {
                     source = "AdminHome.jsp";
              }

       } catch (Exception e2)    {
              e2.printStackTrace();
       }
       return source;
  }
}
```

The code for NoLogin.jsp is as follows:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>NotLogin</title>
</head>
<body>
<center><h3>You have logged out from the system.</h3> <br />
      <h3>Please Sign in again to continue !!!</h3><br /><br /> </center>

<div style="text-align: center"><a
href="<%=request.getContextPath()%>/Login.jsp">Login</a></div>
</body>
</html>
```

Execution Flow Case 2:

To solve the case 2, the system needs to **remember the last page displayed** and that page is **redisplayed in the browser** when the user **tries to log in from a different window of the browser** while one window is already login to the system.

To understand the case (2), we explain with the previous process of adding new book as example. When the user has login to the system and inserting new book in one window and another login attempts in another window of the same browser, the current user session is already opened so that another new session should not be started without proper logout. Therefore the system needs to remember the last page displayed and that page is redisplayed in the browser. This is done by saving the page as String objects in the `doPost()` method of the `LibraryServlet` class and is as follows:

```
public void doPost(HttpServletRequest request,
        HttpServletResponse response)throws IOException,
                                            ServletException {
LastPage = run(request, response);
if (!notLoggedIn(request)) {
   request.setAttribute("lastpage", LastPage);
}
System.out.println("Process output: " + LastPage);
request.getRequestDispatcher(LastPage).forward(request, response);

}
```

In above code, the `run()` method returns ***jsp*** page name as a String object and is saved in the attribute named "`Lastpage`" of the **session**. In such a way, the system remembers the last page displayed and that page is redisplayed in the browser when he/she come back to the system.

To redisplay the last page when user login again in new window, the run() method in LoginServlet class check whether notLoggedIn(request) and redirect to the lastpage. The following code snip as follows.

```
//check there is already logged in user,
//if so redirect to the last page visited
if (!notLoggedIn(request)) {
      response.sendRedirect(LastPage);
      return LastPage;
}
```

The updated LibraryServlet class is shown below:

```
package com.mvc.controller;

import java.io.IOException;
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 javax.servlet.http.HttpSession;

import com.mvc.model.Library;

/**
 * Servlet implementation class LibraryServlet
 */
@WebServlet("/LibraryServlet")
public abstract class LibraryServlet extends HttpServlet {

      private static final long serialVersionUID = 1L;
```

```
        protected Library library;
        public String LastPage;

        public void init() {
          if (library== null)
               library = new Library();
        }

    public LibraryServlet() {
        super();
    }

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

               doPost(request, response);
        }

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

           LastPage = run(request, response);

           if (!notLoggedIn(request)) {
               request.setAttribute("lastpage", LastPage);
           }
           System.out.println("Process output: " + LastPage);
           request.getRequestDispatcher(LastPage).forward(request, response);
        }

        public boolean notLoggedIn(HttpServletRequest request) {
        //checks user who may have invoked a command without actually login in.
        //return true if and only if the user has not currently logged in.
return request.getSession(false).getAttribute("currentUser") == null;
        }

        public String noLoginErrorMessage() {
        // This method simply generates an HTML page that displays 'Not logged
in' and
        // supplies a link to the log in screen.
        // when a person who has not logged in attempts to execute a command.
               return "NoLogin.jsp";
        }

        public abstract String run(HttpServletRequest request,
HttpServletResponse response);

}
```

The updated LoginServlet class is shown below:

```
package com.mvc.controller;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
…
import com.mvc.bean.LoginBean;
import com.mvc.dao.LoginDao;

/**
 * Servlet implementation class LoginServlet
 */
@WebServlet("/LoginServlet")
public class LoginServlet extends LibraryServlet {
        private static final long serialVersionUID = 1L;

    public LoginServlet() {
```

```
        super();
    }
      /**
       * Abstract run() method implementation
       */
    @Override
    public String run(HttpServletRequest request, HttpServletResponse
response) {

      try {
 //check there is already logged in user, if so redirect to the last page
visited
             if (!notLoggedIn(request)) {
                   response.sendRedirect(LastPage);
                   return LastPage;
             }

             String userid = request.getParameter("userid");
             String password = request.getParameter("password");

             //checks whether the user is a superuser or member
             LoginBean loginBean = new LoginBean();
             loginBean.setUserId(userid);
             loginBean.setPassword(password);

             LoginDao loginDao = new LoginDao();
             loginBean = loginDao.authenticateUser(loginBean);

             //some code not shown

      } catch (Exception e2)    {
                  e2.printStackTrace();
       }
       return LastPage;
  }
}
```

Execution Flow Case 3:

To solve the case 3, when the user overwrites the current page by visiting some other site and wants to come back to the library system or when a user goes back to main menu from the middle or at the end of a command, the system remembers the user id and password in the attribute "currentUser" and login location is also remembered in the attribute "location" and the type of user is also remembered in the attribute "UserType" so that these values are read from the session objects to determine what the menu should display.

The code for the saving login user information is described in `run()` method of the `LoginServlet`, some code snip is given below.

```
   @Override
   public String run(HttpServletRequest request, HttpServletResponse response)
{

    //some code not shown

     String userid = request.getParameter("userid");
     String password = request.getParameter("password");

   //checks whether the user is a superuser or member
     LoginBean loginBean = new LoginBean();
     loginBean.setUserId(userid);
     loginBean.setPassword(password);

     LoginDao loginDao = new LoginDao();
     loginBean = loginDao.authenticateUser(loginBean);
```

```
     if(loginBean.getRole().equals("Admin")) {
      //Creating a session and setting session attribute
       HttpSession session = request.getSession();
       session.setAttribute("UserType", "Superuser");
       session.setAttribute("currentUser", loginBean);
       session.setAttribute("userName", loginBean.getUserName());
       session.setAttribute("errMessage", "");

      //Check for access location
       String ipAddress = request.getRemoteAddr();
       System.out.println("IPAddress: " + ipAddress);
       if (loginDao.libraryInvocation(ipAddress)) {
            session.setAttribute("location", "Library");
       } else {
            session.setAttribute("location", "Outside");
            session.setAttribute("errMessage", "Please Login from Library
Terminal !!");
       }
       LastPage = "AdminHome.jsp";
       }
     //Check in the Ordinary Member
       else if(loginBean.getRole().equals("Member")) {
      //Creating a session and setting session attribute
       HttpSession session = request.getSession();
       session.setMaxInactiveInterval(10*60);
       session.setAttribute("UserType", "Member");
       session.setAttribute("currentUser", loginBean);
       session.setAttribute("userName", loginBean.getUserName());
       session.setAttribute("errMessage", "");

      //Check for access location
       String ipAddress = request.getRemoteAddr();
       System.out.println("IPAddress: " + ipAddress);
       if (loginDao.libraryInvocation(ipAddress)) {
         session.setAttribute("location", "Library");    }
       else {
         session.setAttribute("location", "Outside");
         session.setAttribute("errMessage", "To IssueBook, Please Login from
Library Terminal !!");
       }
       LastPage = "MemberHome.jsp";
      }

   // some code not shown
}
```

5.5 Removing Book, Issue Book, Renew Book, etc.

The execution of sequence of this removing book is shown in Figure. 7.30 (a) standalone process and (b) web-based process. Removing book from library needs to check if the book is ***issued*** or has ***hold***, it **cannot be removed from library**.

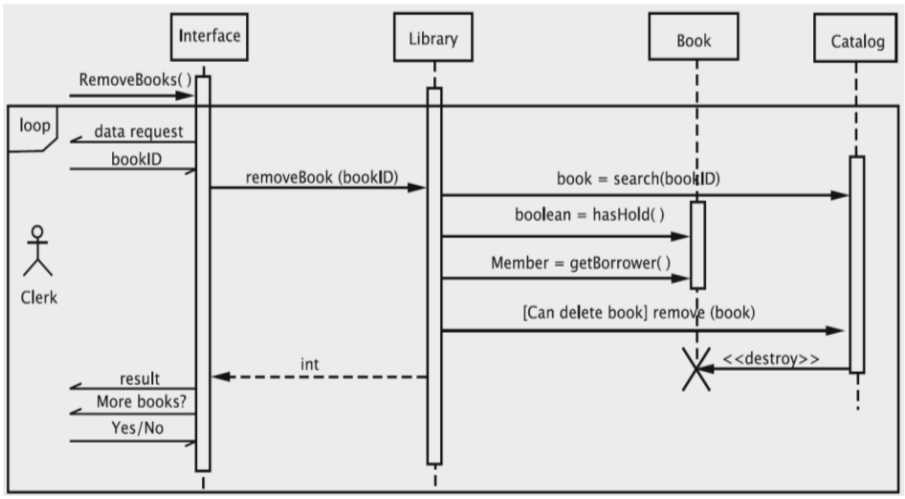


Figure 7.30(a): Sequence diagram for removing books

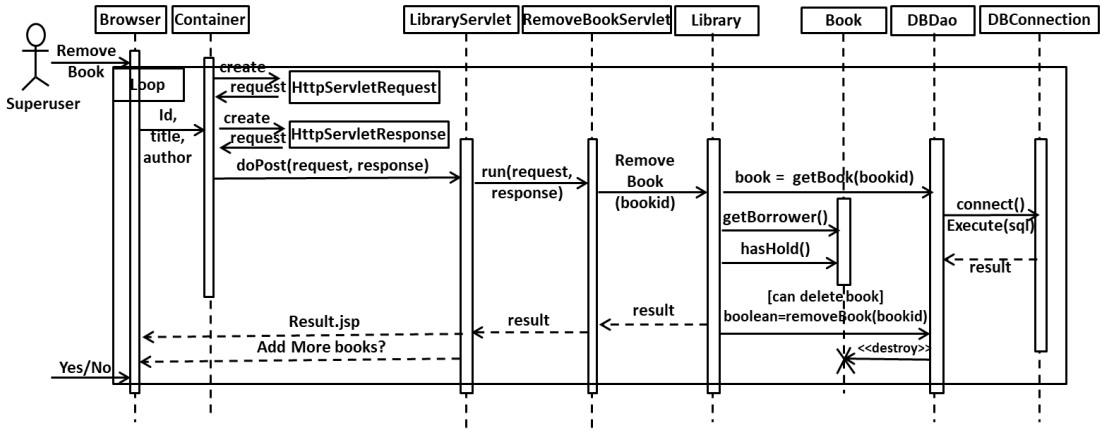


Figure 7.30 (b): Sequence diagram for removing book

The code for some other commands (returning a book, adding a book, adding a member, and processing holds) are quite similar and warrants no further explanation. The other library process such as removing book, issue book, renew book, place a hold are left as exercises for students. Student tries their best to code those process to practice the web application handling between browser and server process.

SUMMARY

The web application architecture is the blueprint for **how different components of a web application, such as databases, applications, and middleware systems, interact with each other and communicate**. It specifies the protocol for exchanging data over the web, ensuring that **both the client-side and server-side can understand and process it**. An **architecture for a web-based application** typically consists of **three main parts**:

(1) **Web Browser**: This is **a client-side component** that enables users to view, interact with, and access web content. It sends an HTTP request to the web server and displays the results for the user.

(2) **Web Server**: This is **a back-end component** that receives requests from the client side and processes them using business logic. It retrieves data from the database and transfers it to the web browser for display.

(3) **Database Server**: This component **stores and retrieves data** in a structured and efficient way, responding to user requests and providing the necessary data for display on the user interface.

Each layer in web application architecture works independently, making it simple to manage, scale, and develop the architecture simultaneously. A well-designed web application architecture is essential for the **performance, scalability, and maintainability of a web application**. It helps to create an application that can easily adapt to changing business needs, handle high levels of traffic, and provide a seamless user experience.

There are various web application architectures, depending on software development and deployment patterns, it can be categorized. Some architecture are monolithic architecture, micro service architecture, container architecture, serverless architectural.

In this lecture we introduce the **MVC architecture** which is a container architecture that is a software architecture pattern to develop a dynamic web application in **a standard manner**. MVC pattern makes **easy to develop JEE application**. Our web based library system is just to understand necessary facts to develop client-server based environment application. In this web app example, we have introduced the session for user, connection to database, application business logic and layer development using MVC pattern.

For web application, the web app system needs to handle user authentication, access control, security and performance issues. Meaning to protect, secure data and maintaining the security.

For the development technology, as we mentioned in this lecture, the Java Servlet technology is just one of the tools available for creating web-based systems. PHP is a scripting language that usually runs on the server side. It can have HTML code embedded into it and outputs web pages. ASP.NET is another competing scripting technology from Microsoft for building web-based applications. JSP is similar to PHP and ASP, the difference being that we intersperse Java code with HTML code to create dynamic web pages. Other technologies such as Ruby on Rails (RoR) are also available.

TheWorldWideWeb Consortium develops technologies for the utilization of the web. This includes specifications for HTML and HTTP. The reader is encouraged to take a look at their site at http://www.w3.org/ to get an overview of the work provided by that group.

********** End of lecture 7 **********

The more that you read, the more things you will know.

The more that you learn, the more places you will go.

Dr. Sesus