## ****TABLE OF CONTENTS****

|  |  |  |
| --- | --- | --- |
| **C.NO** | **TITLE** | **PAGE NO** |
|  | ****ABSTRACT**** | **II** |
| ****1**** | **INTRODUCTION** |  |
| ****2**** | ****Sytem Architecture**** |  |
| ****3**** | **Features** |  |
| ****4**** | **UI/UX** |  |
| ****5**** | Security |  |
| ****6**** | ****Database Schema Design**** |  |
| ****7**** | ****Servlet Details (Business Logic Layer)**** |  |
| ****8**** | ****JSP Pages (Presentation Layer)**** |  |
| ****9**** | ****DAO Layer (Data Access Object)**** |  |
| ****10**** | ****Models (JavaBeans / POJOs)**** |  |
| ****11**** | **References** |  |

**ABSTRACT**

**ABSTRACT**

The development of The **Bus Reservation System** involved implementation of both front-end and back-end components. All system bus processes are based on a dynamic web application system which is developed using **Java Servlets** and **JSP** as the main programming language supporting a SQLite database for maintaining the system bus data and the usability of HTML, CSS and Java script at the front-end.

Delivered convenience for the **users** and **administrators** of the system for managing the issues related to bus reservations along with that ensures level of security and trust. It is a **user**-**centered** application that automates the inefficiencies found in the traditional manual ticketing systems designed to withstand the growing digital era.

The users of the system (i.e. the passengers) are allowed to **register**, **login** to the system, select from the list of **available buses**, **book** and **check bookings**. In an Administrator level, the manages have extra privileges to control bus users as well as the settings of the system from a secured interface. The users also will be design if not all the features of the system will be put into consideration using **MVC strcuture** such that it is modular and maintainable.

Booking and ticketing systems have to provide a formidable challenge in **order to enhance overall passenger experience**. Session based control system was designed but is individual step to secure web application frameworks and restricted areas on a global scale are achieved thus making Domain sensitive edit web page and document layout applications such as ticket details on-power other functions.

**INTRODUCTION**

**Chapter 1**

**INTRODUCTION**

## ****1.1 Overview Bus Reservation System****

The **Bus Reservation System** is a dynamic and secure web-based application developed to simplify the process of reserving and managing bus travel tickets. The system targets both **end-users (passengers)** and **administrators** by offering tailored interfaces and access privileges. Users can register, log in, search for available buses by route and date, book seats, and view their booking history. On the admin side, functionalities include managing bus data, overseeing bookings, modifying user details, and handling system settings from a central dashboard.

This system eliminates the need for manual ticketing processes, offering a streamlined and automated experience for transport companies and their customers. It ensures real-time data access, improves operational speed, and enhances accuracy, making the travel experience more convenient and transparent for everyone involved.

### ****Technologies Used****

This project uses a full-stack Java EE (Jakarta EE) web development approach combined with open-source technologies:

**Frontend:**

**HTML5 / CSS3:** For responsive page layout and styling

**JavaScript:** For basic client-side interactivity

**JSP (JavaServer Pages):** For rendering dynamic content and server-side logic

**Backend:**

**Java Servlets:** Handle HTTP requests, business logic, and session management

**JDBC (Java Database Connectivity):** Used for direct communication with the SQLite database

**Apache Tomcat:** Acts as the web server and servlet container

**Database:**

**SQLite:** Lightweight, serverless relational database to manage persistent data like user records, buses, and bookings

**Architecture:**

**MVC (Model-View-Controller):** Ensures separation of concerns for maintainability and scalability

### ****1.2 Existing System****

In most traditional setups, **bus reservations are handled manually or through outdated desktop applications**, leading to several operational inefficiencies and limitations:

Passengers must physically visit counters or rely on third-party agents for bookings

No real-time seat availability updates — increases chances of overbooking

No centralized system for managing user profiles, cancellations, or transaction records

Difficulty in tracking data history or generating reports

Admin tasks like bus management, user handling, and analytics are highly manual

Lack of proper user authentication or session security

Poor UI/UX experience for end-users; mobile-unfriendly interfaces

Such systems lack scalability, security, and adaptability — especially for businesses aiming to modernize or grow.

### ****1.3 Proposed System****

The **proposed Bus Reservation System** addresses these limitations through a modern, web-based solution that combines **Java (Servlets & JSP), JDBC, and SQLite**, designed using the **MVC architecture**.

**Key Advantages:**

**Online Booking:** Users can register, login, search, and book tickets seamlessly

**Real-time Seat Availability:** Instant updates to available seats after booking

**Role-Based Access:** Separate dashboards for users and admin with dedicated privileges

**Data Centralization:** All data (buses, bookings, users) managed via one SQLite DB

**Security Features:** Secure login, session management, and SQL injection protection using Prepared Statements

**Modular Design:** Easy to maintain and extend (e.g., adding payment gateway or 2FA later)

**User-Centric UI:** Clean layout with HTML/CSS and responsive interactions

**Admin Control Panel:** Admins can manage buses, monitor users, and tweak settings easily

**Optional Features:** Email confirmation, PDF ticket generation, and mobile-friendly extension support

## ****System Architecture****

## ****Chapter 2****

****System Architecture****

The Bus Reservation System follows the **Model-View-Controller (MVC)** architecture pattern, promoting clear separation of concerns, maintainability, and scalability.

### ****2.1 MVC Pattern Overview****

**Model (M)**: Handles all data-related logic and interactions with the database. For example, User.java, Bus.java, Booking.java.

**View (V)**: The presentation layer, built with **JSP + HTML/CSS**, which interacts with users (e.g., login.jsp, home.jsp, adminDashboard.jsp).

**Controller (C)**: Servlets act as controllers that handle client requests, process input, and determine appropriate view rendering (e.g., LoginServlet, TicketBookingServlet).

**Benefits of MVC:**

Cleaner code structure

Easier testing and debugging

Easy to scale or replace components independently

⚠️ Note: The system is designed to run on low-resource devices and is optimized for edge deployment. GPU is optional unless heavy retraining is needed.

**2.2 Project Folder Structure**

BusReservationSystem/

│

├── src/

│ ├── servlets/

│ │ ├── LoginServlet.java

│ │ ├── RegisterServlet.java

│ │ ├── TicketBookingServlet.java

│ │ └── AdminServlet.java

│ ├── dao/

│ │ ├── UserDAO.java

│ │ └── BusDAO.java

│ └── model/

│ ├── User.java

│ ├── Bus.java

│ └── Booking.java

│

├── WebContent/

│ ├── login.jsp

│ ├── register.jsp

│ ├── home.jsp

│ ├── adminDashboard.jsp

│ ├── viewUsers.jsp

│ └── error.jsp

│

└── database/

└── bus.db (SQLite)

### ****2.3 Component Interaction Flow****

Here’s how different components interact with each other:

**User sends request** (e.g., login, register, book) via a JSP form.

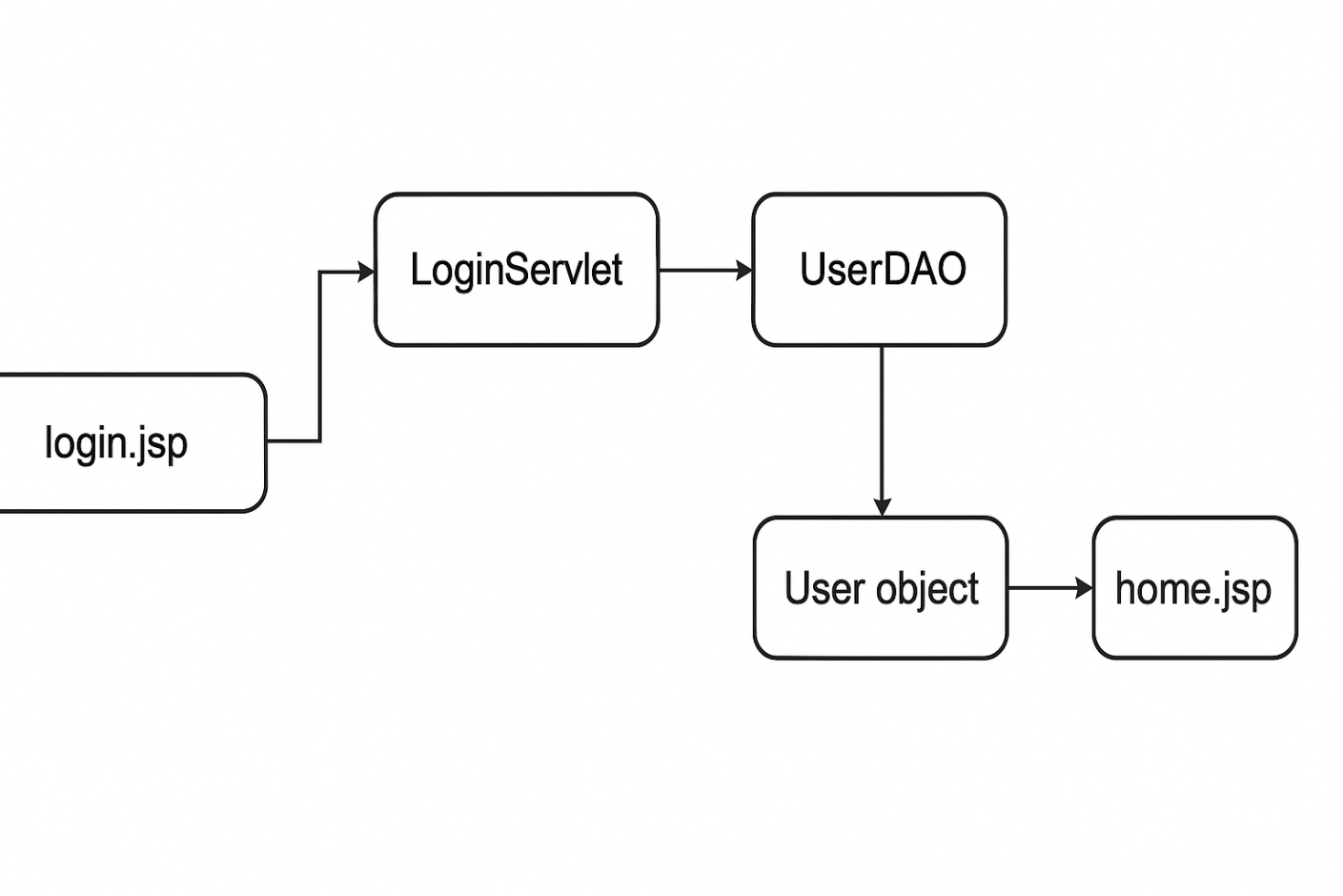
**Servlet handles logic** – validates input, interacts with the DAO.

**DAO layer accesses the database** – fetches/inserts user or booking data.

**Model object is created** – holds relevant data (e.g., User or Bus instance).

**Forward/redirect to JSP** – shows user appropriate response or view.

**Example Flow (Login):**  
login.jsp ➝ LoginServlet ➝ UserDAO ➝ SQLite DB ➝ User object ➝ home.jsp



**fig 2.1 Flow Diagram**

**Features**

**Chapter 3**

**Features**

The Bus Reservation System provides a set of core functionalities that cater to both regular users and admin users, ensuring a seamless and efficient bus booking experience. These features are designed to offer ease of use, security, and flexibility for different roles.

**3.1 User Registration and Login**

· **User Registration**: Users can create an account by providing a username, email, and password. A confirmation email can be sent to verify the email address (optional).

· · **Login**: After registration, users can log in using their credentials (username and password). If the credentials match, they gain access to the system.

· · **Error Handling**: If an error occurs (e.g., invalid credentials or missing registration fields), appropriate error messages are displayed.

·

**3.2 Role-Based Access (User/Admin)**

· **User Role**: Regular users have access to the booking system, where they can:

View available buses.

Book tickets based on source, destination, and travel date.

View and cancel their own bookings.

· **Admin Role**: Admins can manage the entire system, including:

Viewing and deleting users.

Modifying bus details (e.g., bus name, travel schedule, fare).

Managing bookings (viewing, canceling, and approving them).

Configuring system settings (optional).

### ****3.3 Ticket Booking System****

**Bus Availability**: Users can search for available buses based on their travel requirements (source, destination, and date)

**Booking Tickets**: Once a user finds a suitable bus, they can proceed to book tickets. The system ensures that available seats are properly managed and updates the available seats count.

**Booking Confirmation**: After successfully booking a ticket, users receive a booking confirmation, which can be viewed and printed.

**Fig 3.2.1: Data Flow Diagram**

### ****3.4 Admin Dashboard****

The admin dashboard is a central hub for managing the system:

**Manage Users**: Admins can view the list of registered users, delete users, or update their roles.

**Manage Bookings**: Admins can view all bookings, cancel bookings, or modify booking statuses.

**System Settings**: Admins can configure system-wide settings such as bus details, fare updates, or travel dates.

### ****3.5 Email Confirmation (Optional)****

**Email Notifications**: Upon booking a ticket or performing other significant actions, users may receive email notifications to confirm the action (e.g., ticket booking, password reset). This feature can be integrated for added user convenience and security.

### ****3.6 PDF Ticket Generation (Optional)****

**PDF Ticket**: After a successful booking, users can download a PDF version of their ticket containing:

Booking ID

User details (name, email)

Bus details (name, departure, arrival, fare)

Seat number

### ****3.7 Logout Functionality****

**Logout**: Both users and admins can log out of the system at any time. This ends the session, ensuring that users cannot access restricted pages once logged out.

**Session Management**: Once a user logs out, the session is destroyed to prevent unauthorized access.

## UI/UX

## Chapter 4

**UI/UX**

**Responsive Design**: The application uses responsive web design principles, ensuring that the UI is adaptive to different screen sizes, including desktop, tablet, and mobile devices. This is achieved by using **CSS media queries** to adjust layout components, font sizes, and button styles based on screen width.

Example of a basic CSS media query:

/\* For screens wider than 768px (tablets and above) \*/

@media screen and (min-width: 768px) {

body {

font-size: 16px;

}

}

/\* For smaller screens like mobile phones \*/

@media screen and (max-width: 767px) {

body {

font-size: 14px;

}

}

**Flexbox and Grid Layouts**: These modern CSS layout systems help in creating flexible and responsive designs. Flexbox is used for aligning and distributing space among items in a container, while CSS Grid allows for complex layouts with rows and columns.

Example of a Flexbox layout for a navigation menu:

.navbar {

display: flex;

justify-content: space-between;

padding: 10px;

}

.navbar a {

text-decoration: none;

color: white;

padding: 10px;

}

**CSS Variables**: To improve maintainability and make the styling process more efficient, CSS variables are used to store values such as colors, font sizes, and spacing. This allows for easy modification of the design in one place.

Example of CSS variables:

:root {

--primary-color: #00adb5;

--secondary-color: #e74c3c;

--font-size: 16px;

}

body {

font-size: var(--font-size);

background-color: var(--primary-color);

}

.button {

background-color: var(--secondary-color);

}

### ****4.2 JSP Integration****

JavaServer Pages (JSP) is used to create dynamic web pages, enabling the display of user-specific data and interaction with the backend logic.

**Dynamic Content**: JSP pages are used to generate dynamic content by embedding Java code into HTML. This is essential for showing personalized information, such as a logged-in user's booking history or admin data.

Example of displaying a user’s name dynamically:

<h2>Welcome, <%= user.getUsername() %></h2>

**Form Handling**: JSP pages are also used to handle forms. For instance, the login form sends data to the server, where it is processed to authenticate the user.

Example of a login form:

<form action="login" method="post">

<label for="username">Username</label>

<input type="text" id="username" name="username" />

<label for="password">Password</label>

<input type="password" id="password" name="password" />

<input type="submit" value="Login" />

</form>

### ****Navigation Flow****

**Easy Navigation**: A key aspect of UX design is ensuring smooth and intuitive navigation throughout the application. A consistent navigation bar allows users to access various parts of the system like home, ticket booking, and user settings. For admin users, an additional dashboard for managing users and bookings is available.

Example of a navigation bar (HTML + CSS):

<div class="navbar">

<a href="home.jsp">Home</a>

<a href="booking.jsp">Book Tickets</a>

<a href="logout.jsp">Logout</a>

</div>

**Breadcrumb Navigation**: For more complex pages, such as booking history or admin dashboards, breadcrumb navigation can be implemented to show users where they are within the system.

Example of a breadcrumb trail:

<div class="breadcrumb">

<a href="home.jsp">Home</a> > <a href="mybookings.jsp">My Bookings</a>

</div>

### ****4.4 Error Handling and Messages****

**User-Friendly Error Messages**: A major part of UX design is providing users with clear, concise, and helpful error messages. In the Bus Reservation System, users are shown informative messages when they enter invalid input or encounter system errors.

Example of an error handling page:

<div class="error-container">

<h1>404</h1>

<h2>Page Not Found</h2>

<p>Oops! The page you're looking for doesn't exist or has been moved.</p>

<p><a href="<%= request.getContextPath() %>">Go back</a></p>

</div>

**Validation Feedback**: When users input data (e.g., booking details or registration forms), immediate feedback should be provided if any fields are missing or incorrectly filled out.

Example of inline form validation:

<form action="register" method="post">

<label for="username">Username</label>

<input type="text" id="username" name="username" required />

<div class="error" id="username-error"></div>

</form>

### ****4.5 User Experience Enhancements****

**Modal Windows**: For actions like booking confirmation or viewing detailed bus information, modal windows can be used to provide a smooth user experience without requiring page reloads.

Example of a basic modal using HTML, CSS, and JavaScript:

<div id="myModal" class="modal">

<div class="modal-content">

<span class="close">&times;</span>

<p>Booking Confirmed!</p>

</div>

</div>

<script>

var modal = document.getElementById("myModal");

var span = document.getElementsByClassName("close")[0];

span.onclick = function() {

modal.style.display = "none";

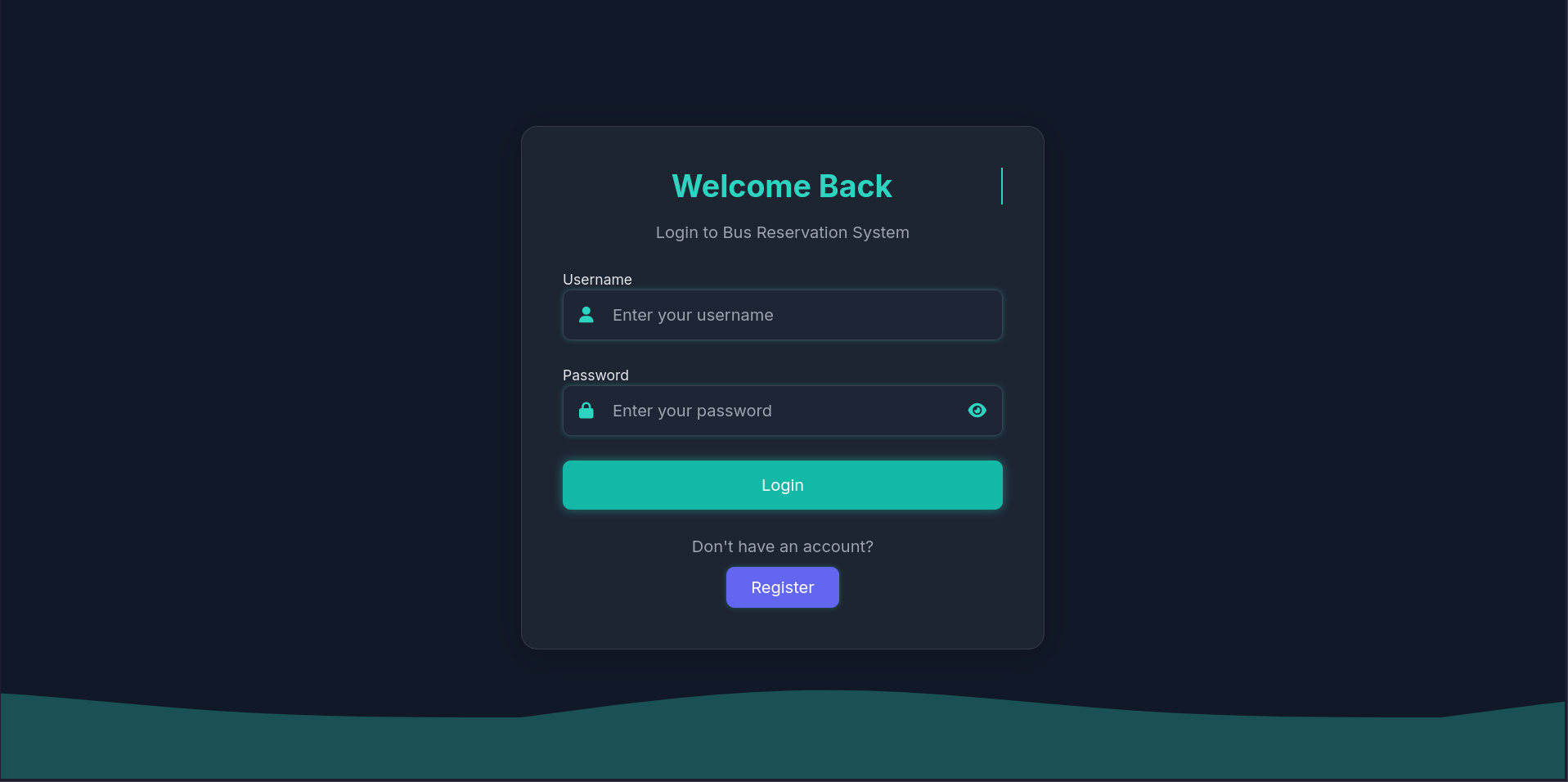
}

</script>

### ****4.6 Final Thoughts on UI/UX****

The UI/UX design in the Bus Reservation System is focused on providing an intuitive and user-friendly experience across all roles (users and admins). By using responsive layouts, enhancing usability with form validations, providing clear feedback to users, and ensuring easy navigation, the system offers a seamless interaction. The design also emphasizes flexibility by using modern CSS techniques like Flexbox, CSS Grid, and media queries, along with dynamic content rendering via JSP.

Let me know if you'd like to proceed to the **Security** section or need further elaboration!



## fig 4.1 Login page

## 

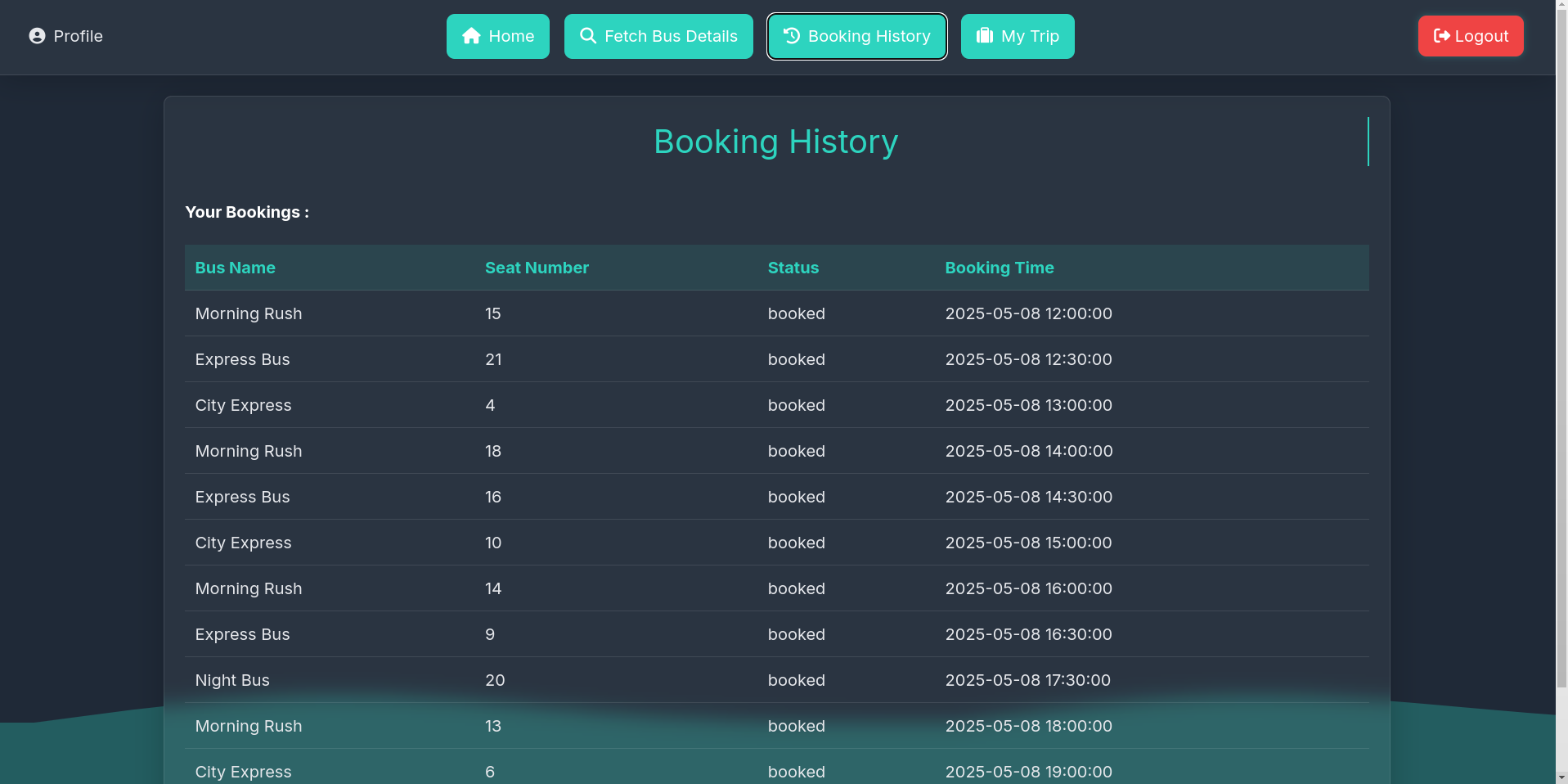
## fig 4.2 Register page

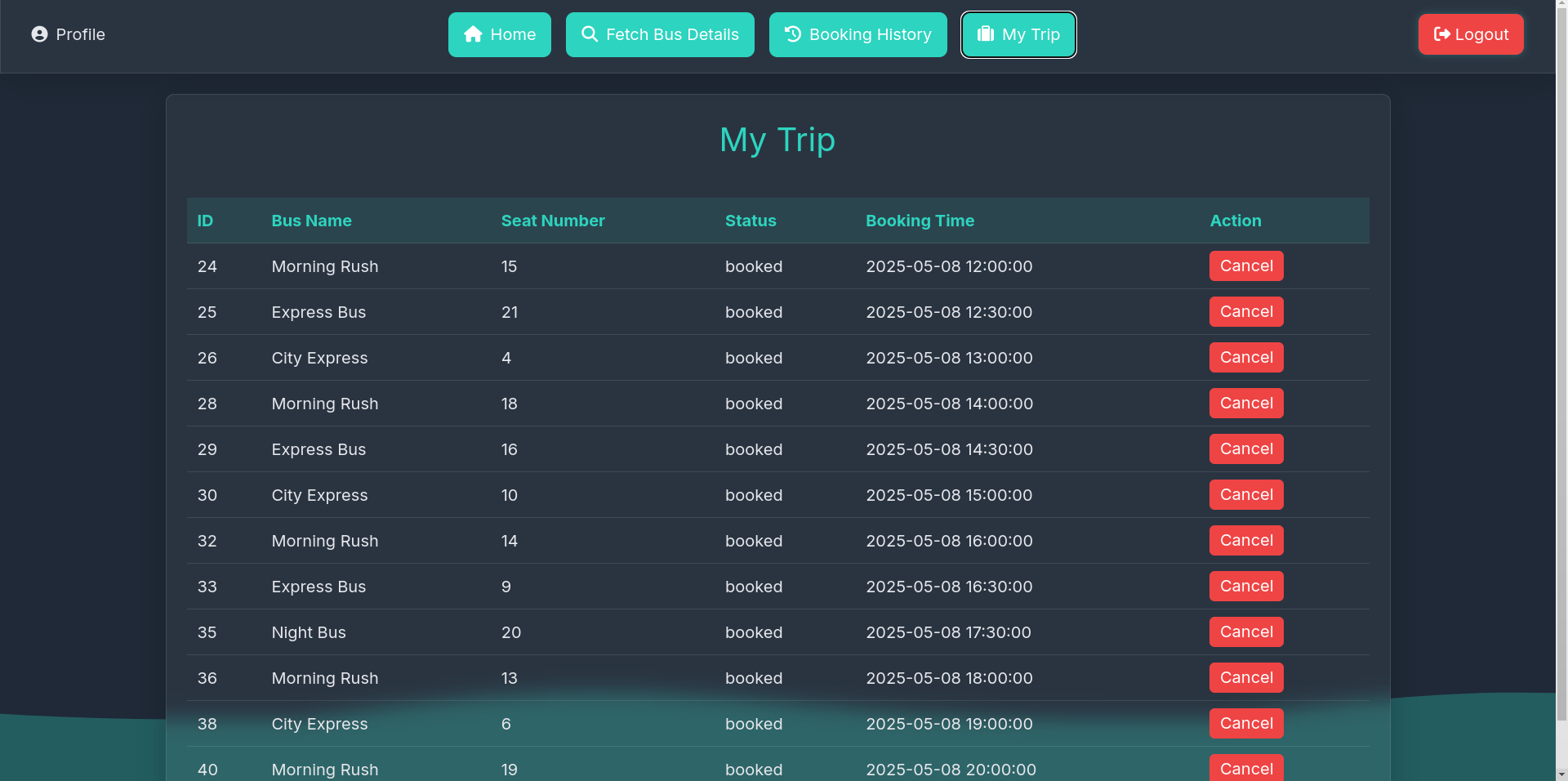
## 

## fig 4.3 Home page

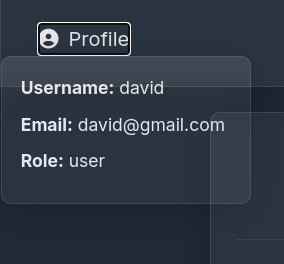
## 

**fig 4.4 Fetch Bus Details page**

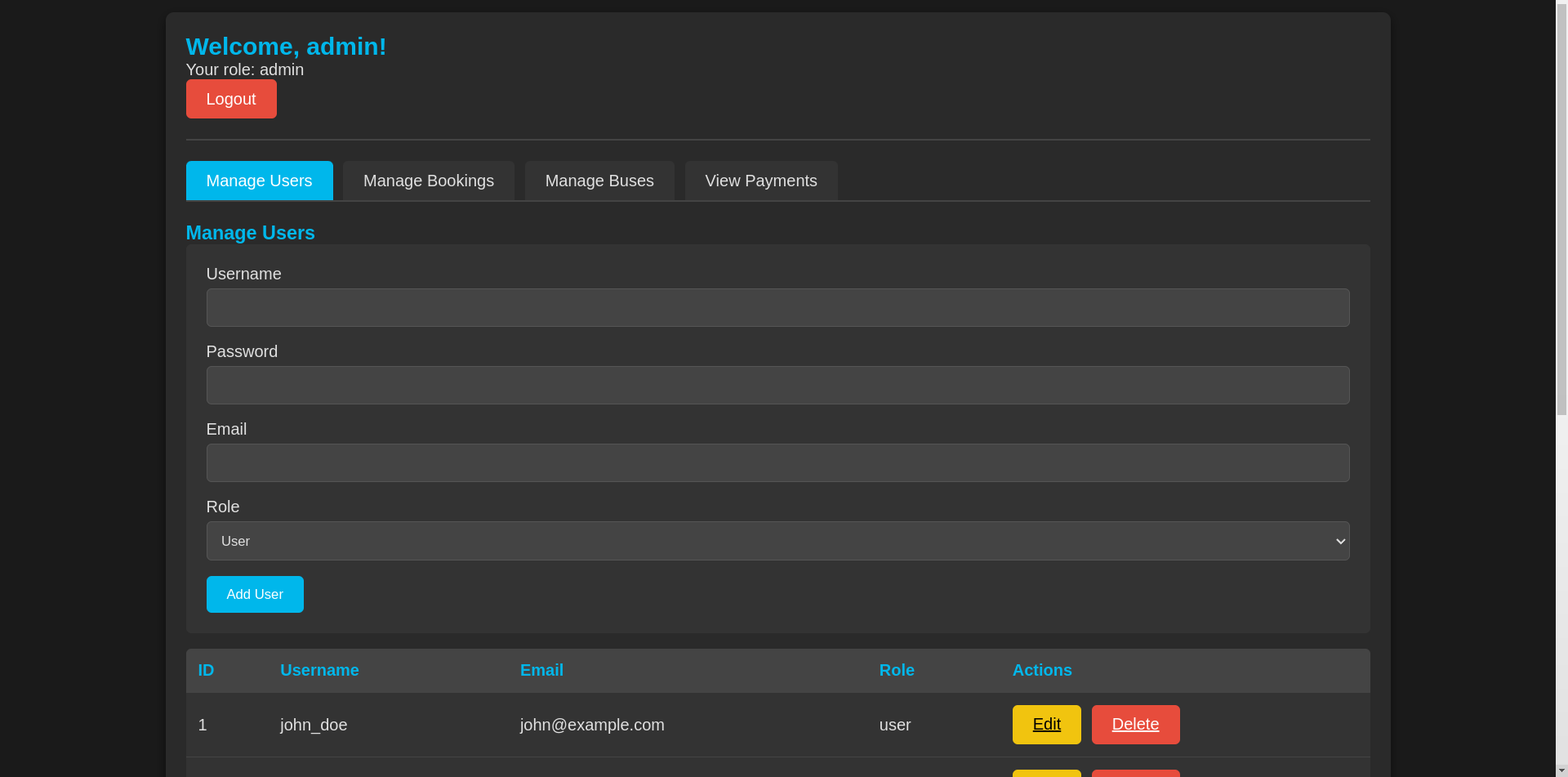
**fig 4.5 Ticket Booking page**



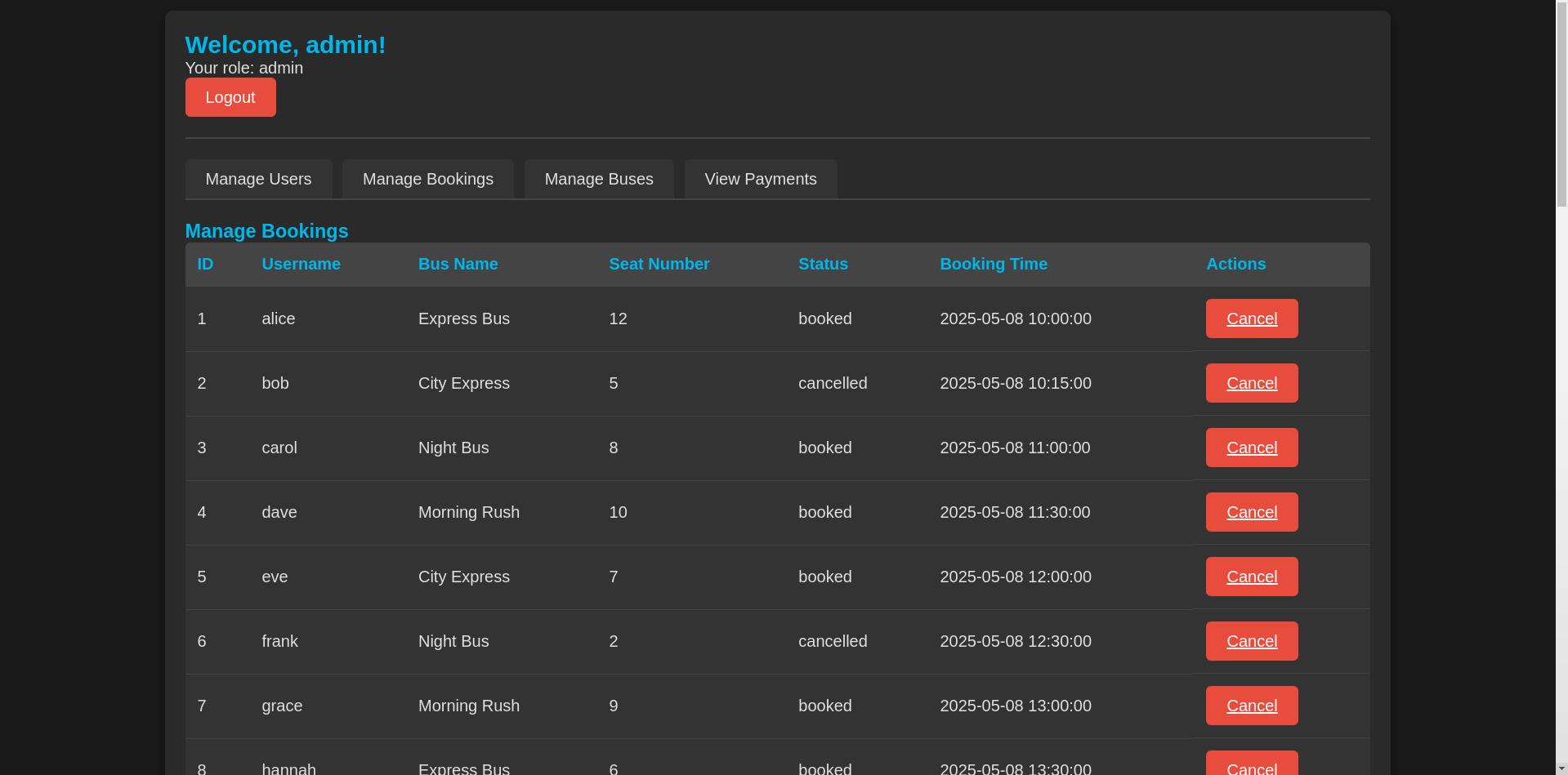
**fig 4.6 Ticket Cancel page (MyTrips**



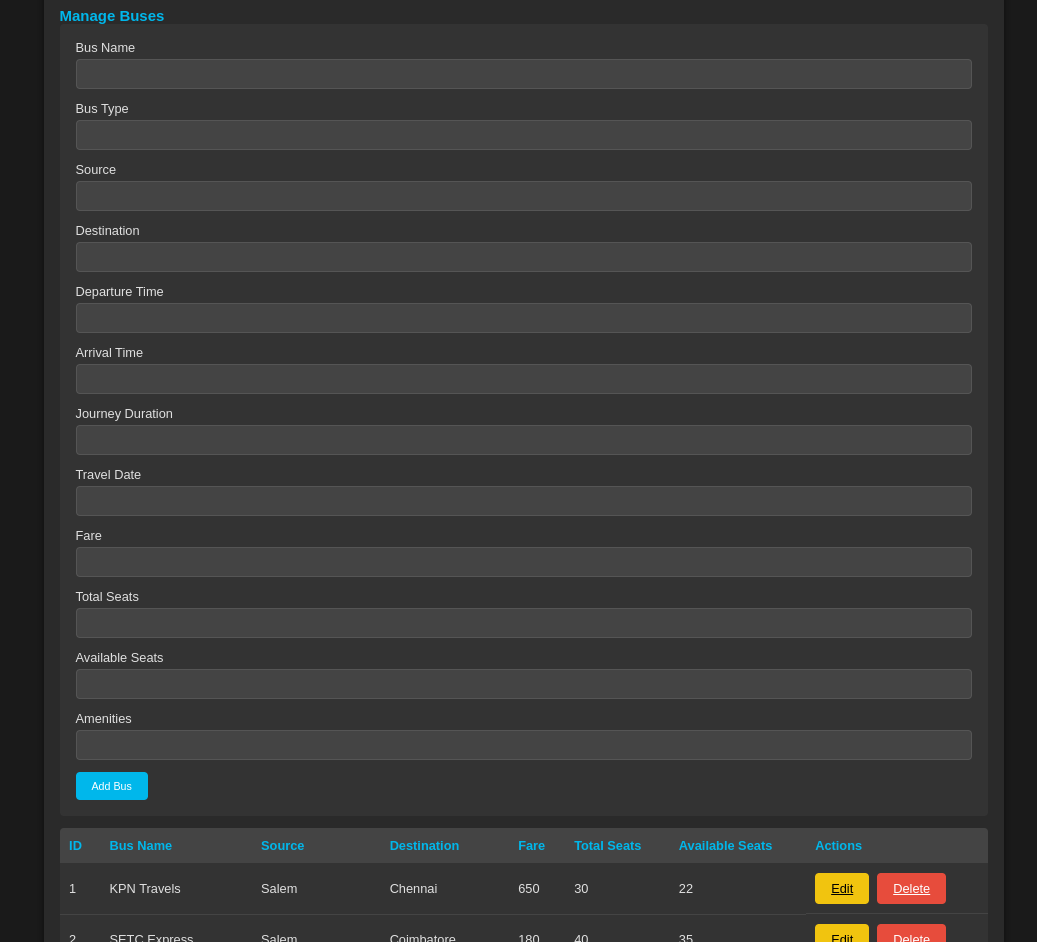
**fig 4.7 Profile page**



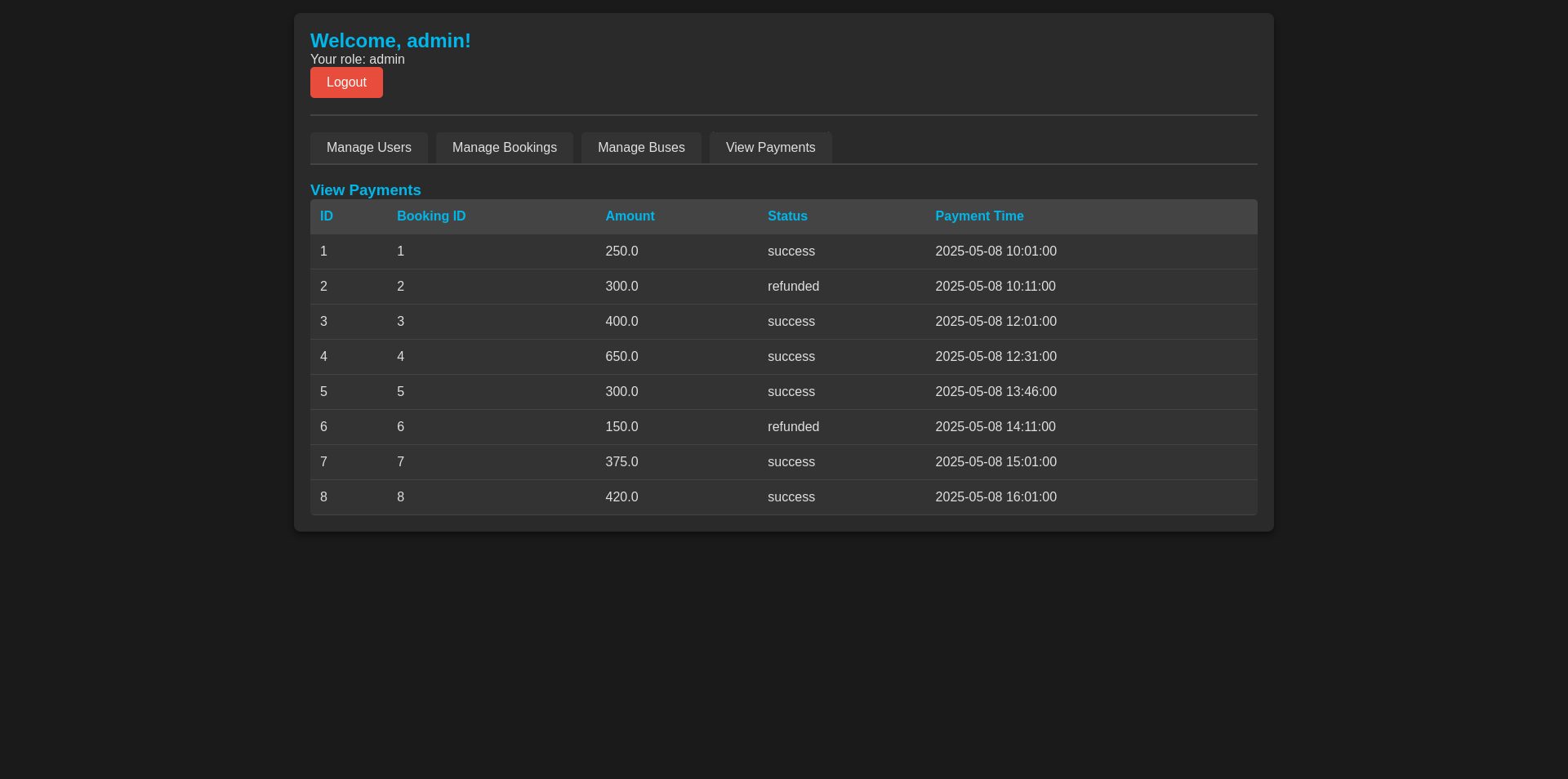
**fig 4.8 Admin page (Manage Users)**



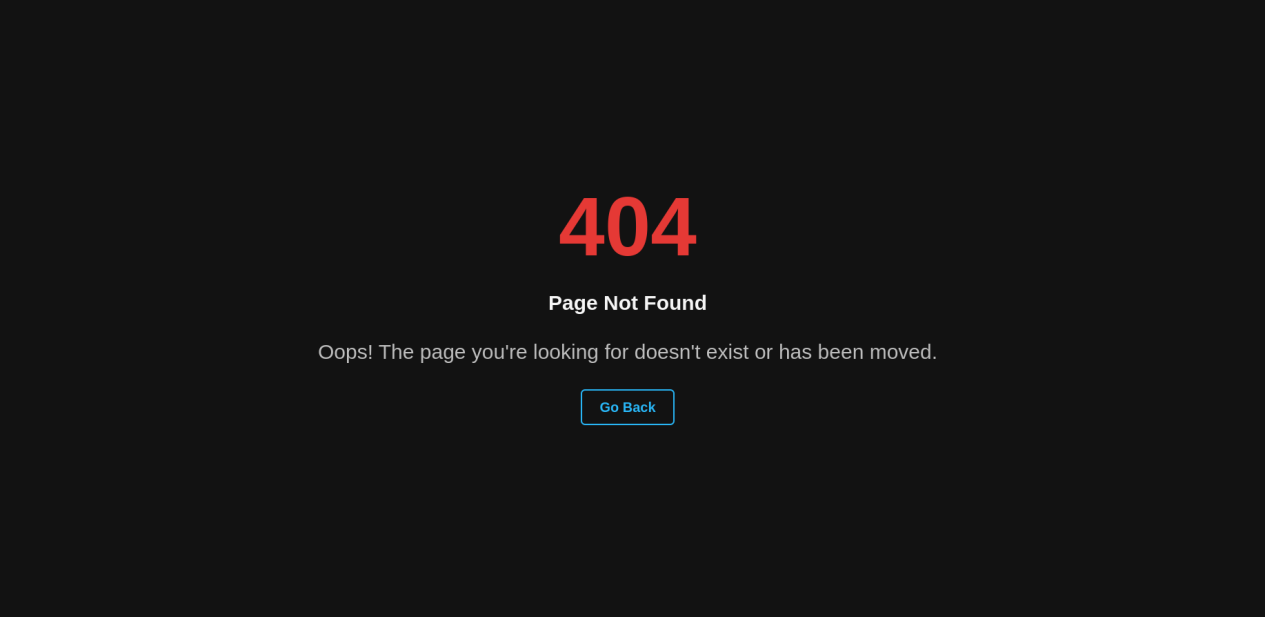
**fig 4.9 Admin page (Manage Bookings)**



**fig 4.10 Admin page (Manage Buses)**



**fig 4.11 Admin page (View Payments)**



**fig 4.12 Error page**

## Security

## Chapter 5

**Security**

Security is a fundamental aspect of any web application, particularly for systems handling sensitive data like user information, booking details, and payment-related processes. For the **Bus Reservation System**, ensuring the integrity and confidentiality of user data is paramount. This chapter will cover various security measures implemented in the system, focusing on session management, authentication, authorization, input validation, SQL injection protection, password handling, and access control.

### ****5.1 Session Management****

Session management ensures that user interactions with the system are securely tracked during their visit. Proper session handling is crucial to protect against session hijacking, fixation, and other attacks.

**Session Initialization**: When a user logs in, the system creates a session that is associated with their unique session ID. This session is used to store user-related data such as username, role (user or admin), and login status. A session ID is typically stored in a browser cookie, with the HttpOnly flag set to prevent JavaScript access to the cookie.

Example of session creation in a servlet:

* HttpSession session = request.getSession();
* session.setAttribute("username", user.getUsername());
* session.setAttribute("role", user.getRole());

**Session Expiration**: Sessions are set to expire after a period of inactivity. This helps prevent unauthorized access if a user leaves the session open on a public computer. The session timeout is configured in the web.xml configuration file.

Example of session timeout configuration in web.xml:

<session-config>

<session-timeout>30</session-timeout> <!-- Timeout in minutes -->

</session-config>

**Session Fixation Protection**: To protect against session fixation attacks, where an attacker forces a victim to use a predetermined session ID, the system ensures that a new session ID is generated after successful login.

Example of regenerating session ID:

HttpSession session = request.getSession();

session.invalidate(); // Invalidates the current session

session = request.getSession(true); // Creates a new session

### ****5.2 Authentication & Authorization****

Authentication and authorization are the cornerstone of web application security. Authentication verifies the identity of a user, while authorization ensures that the user has the right to perform specific actions.

**Authentication**: The system implements a login mechanism where users authenticate using their username and password. Upon successful authentication, a session is created to keep track of the logged-in user.

Example of user authentication:

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

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

User user = userDAO.authenticate(username, password);

if (user != null) {

HttpSession session = request.getSession();

session.setAttribute("user", user);

response.sendRedirect("home.jsp");

} else {

request.setAttribute("error", "Invalid credentials");

request.getRequestDispatcher("login.jsp").forward(request, response);

}

**Authorization**: The system implements role-based access control (RBAC) to differentiate between regular users and admins. Admin users are granted additional privileges, such as managing bookings and modifying bus schedules.

Example of role-based access control:

HttpSession session = request.getSession();

User user = (User) session.getAttribute("user");

if (user != null && "admin".equals(user.getRole())) {

// Grant access to admin dashboard

} else {

response.sendRedirect("accessDenied.jsp");

}

### ****5.3 Input Validation****

Input validation ensures that data submitted by users is both correct and safe. It helps prevent security vulnerabilities such as **Cross-Site Scripting (XSS)**, **SQL Injection**, and **Buffer Overflows**.

**Client-Side Validation**: Basic validation is performed on the client side using HTML5 input attributes (e.g., required, type="email", minlength, etc.). This helps catch common user errors before data is submitted to the server.

Example of client-side validation:

<form action="register" method="post">

<input type="text" name="username" required />

<input type="email" name="email" required />

<input type="password" name="password" minlength="8" required />

</form>

### ****5.4 SQL Injection Protection (Prepared Statements)****

SQL injection is one of the most common vulnerabilities that allows attackers to manipulate SQL queries by injecting malicious SQL code. To protect the system from SQL injection, **Prepared Statements** are used.

**Prepared Statements**: Prepared statements separate SQL queries from user input, preventing attackers from injecting malicious SQL code. By using parameterized queries, the system ensures that user input is treated as data, not executable code.

Example of a prepared statement:

String query = "SELECT \* FROM users WHERE username = ? AND password = ?";

PreparedStatement stmt = connection.prepareStatement(query);

stmt.setString(1, username);

stmt.setString(2, password);

ResultSet rs = stmt.executeQuery();

### ****5.5 Password Handling****

Storing plain-text passwords is a major security risk. Therefore, passwords should always be hashed before storage to prevent exposure in case of a data breach.

**Password Hashing**: The system uses a secure hashing algorithm (e.g., **bcrypt** or **PBKDF2**) to hash passwords. This process ensures that even if an attacker gains access to the database, they will not be able to retrieve the original passwords.

### ****5.6 Access Control (Admin vs User)****

Access control ensures that users can only access resources and perform actions for which they are authorized.

**Admin Privileges**: Admin users have additional privileges, such as managing users, modifying bus schedules, and monitoring booking history. These privileges are enforced through authorization checks.

Example of admin check:

if ("admin".equals(user.getRole())) {

// Allow access to admin features

} else {

response.sendRedirect("accessDenied.jsp");

}

**User Privileges**: Regular users can book tickets, view their bookings, and update their personal information. They are not allowed to access administrative pages.

Example of user-specific actions:

if ("user".equals(user.getRole())) {

// Grant access to user booking page

} else {

response.sendRedirect("accessDenied.jsp");

}

### ****5.7 Security Best Practices****

In addition to the measures discussed above, the system implements the following security best practices:

**HTTPS**: The entire system is served over HTTPS to ensure that all communication between the user and the server is encrypted.

**Content Security Policy (CSP)**: The system uses a CSP header to prevent certain types of attacks, like XSS, by controlling which resources can be loaded on the page.

Example of CSP header:

Content-Security-Policy: default-src 'self'; script-src 'self' https://apis.google.com;

### ****5.8 Final Thoughts on Security****

The **Bus Reservation System** employs a range of security measures to protect user data, prevent unauthorized access, and defend against common web vulnerabilities. By using secure session management, strong authentication and authorization mechanisms, input validation, and modern hashing techniques, the system ensures that user data remains safe. Additionally, the system is built with security best practices in mind, including the use of HTTPS, CSRF protection, and a robust content security policy.

This comprehensive approach to security helps mitigate the risk of attacks and provides a reliable and safe platform for users to interact with the system.

****Database Schema Design****

### ****Chapter 6****

## ****Database Schema Design****

The **Bus Reservation System** relies on a well-structured relational database (bus.db) to manage all core functionalities—users, buses, bookings, and payments. This chapter details the schema design, table relationships, and how each part contributes to the system's logic.

**6.1 Overview**

The database consists of four main tables:

| **Table Name** | **Description** |
| --- | --- |
| users | Stores user credentials and roles |
| buses | Holds details about available buses |
| bookings | Records reservations made by users |
| payments | Logs payment details for each booking |

### ****6.2 Table Structures****

#### ****6.2.1**** users ****Table****

This table stores the credentials and identity information of all users (regular users and admins).

CREATE TABLE users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

username TEXT UNIQUE NOT NULL,

password TEXT NOT NULL,

email TEXT UNIQUE NOT NULL,

role TEXT NOT NULL DEFAULT 'user' -- 'user' or 'admin'

);

**Columns:**

id: Auto-incremented unique identifier.

username: Login name, unique per user.

password: Hashed password (recommended in production).

email: Unique email address.

role: Specifies whether the user is a 'user' or 'admin'.

#### ****6.2.2**** buses ****Table****

This table contains information about all buses that are available for booking.

CREATE TABLE buses (

id INTEGER PRIMARY KEY AUTOINCREMENT,

bus\_name TEXT NOT NULL,

source TEXT NOT NULL,

destination TEXT NOT NULL,

departure\_time TEXT NOT NULL,

arrival\_time TEXT NOT NULL,

total\_seats INTEGER NOT NULL

);

**Columns:**

id: Unique bus identifier.

bus\_name: Bus label (e.g., "Sunset Express").

source and destination: Start and end points

departure\_time, arrival\_time: Schedule times.

total\_seats: Total available seats.

#### ****6.2.3**** bookings ****Table****

Handles the core reservation logic by linking users to specific buses.

CREATE TABLE bookings (

id INTEGER PRIMARY KEY AUTOINCREMENT,

user\_id INTEGER,

bus\_id INTEGER,

booking\_time TEXT,

seat\_number INTEGER,

status TEXT DEFAULT 'booked', -- or 'cancelled'

FOREIGN KEY(user\_id) REFERENCES users(id),

FOREIGN KEY(bus\_id) REFERENCES buses(id)

);

**Columns:**

id: Unique booking ID.

user\_id: Foreign key referencing users

bus\_id: Foreign key referencing buses.

booking\_time: Time the booking was made

seat\_number: Which seat was reserved.

status: Booking status (booked or cancelled).

#### ****6.2.4**** payments ****Table****

Each booking may have a payment associated with it.

sql

CREATE TABLE payments (

id INTEGER PRIMARY KEY AUTOINCREMENT,

booking\_id INTEGER,

amount REAL,

payment\_status TEXT DEFAULT 'success',

payment\_time TEXT,

FOREIGN KEY(booking\_id) REFERENCES bookings(id)

);

**Columns:**

id: Unique payment ID.

booking\_id: Foreign key linking to a booking.

amount: Total payment amount.

payment\_status: success or refunded.

payment\_time: Timestamp of the transaction.

### ****6.3 Table Relationships****

Here's how the tables connect logically:

**users ↔ bookings**  
A user can have many bookings (1:N).

**buses ↔ bookings**  
A bus can be booked multiple times (1:N).

**bookings ↔ payments**  
One booking can result in one payment (1:1 in practice).

**ER Diagram :**

users(id) ─┬──▶ bookings(user\_id)

│buses(id) ─┘

bookings(id) ─▶ payments(booking\_id)

### ****6.4 Dummy Data Insights****

Based on your extracted .db dump:

**Users:**

eve\_shadow (user), diana\_admin (admin), charlie\_hacks (user),

bob\_builder (user), alice\_wonder (user), admin (admin), john\_doe (user)

**Buses:**

MidnightRide (CityC → CityE)

SunsetExpress (CityB → CityD)

BeachRunner (CityE → CityA)

MountainCruiser (CityD → CityE)

HighwayStar (CityA → CityD)

NightRide (CityA → CityC)

FastWheels (CityB → CityC)

**Bookings:**

Mixed bookings with a range of timestamps, statuses (booked, cancelled), and seat numbers.

**Payments:**

Status includes success and refunded, corresponding to the booking’s status.

### ****6.5 Indexes & Optimization Suggestions****

To boost query performance:

Add indexes on user\_id, bus\_id, and booking\_id.

Normalize repetitive text fields or use enums where supported.

Use DATE/DATETIME for more manageable date queries (if supported by the JDBC driver).

### ****Servlet Details (Business Logic Layer)****

### ****Chapter 7****

****Servlet Details (Business Logic Layer)****

This chapter explains the **core backend logic** using Java Servlets. These servlets are the bridge between the front-end (JSP/HTML) and the backend (Database/DAO layer). Each servlet handles specific tasks like login, registration, booking, and admin operations.

### ****7.1 Servlet Overview****

| **Servlet Name** | **Purpose** |
| --- | --- |
| LoginServlet | Authenticates users |
| RegisterServlet | Registers new users |
| TicketBookingServlet | Handles ticket bookings |
| AdminServlet | Admin tasks (manage users, buses, etc.) |
| LogoutServlet | Clears session and logs out user |

### ****7.2 LoginServlet****

Handles the authentication process by checking user credentials.

**Sample Code:**

@WebServlet("/login")public class LoginServlet extends HttpServlet {

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

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

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

UserDAO userDAO = new UserDAO();

User user = userDAO.authenticateUser(username, password);

if (user != null) {

HttpSession session = request.getSession();

session.setAttribute("user", user);

if (user.getRole().equals("admin")) {

response.sendRedirect("adminDashboard.jsp");

} else {

response.sendRedirect("home.jsp");

}

} else {

request.setAttribute("error", "Invalid credentials");

request.getRequestDispatcher("login.jsp").forward(request, response);

}

}

}

### ****7.3 RegisterServlet****

Registers a new user and inserts the data into the users table.

@WebServlet("/register")public class RegisterServlet extends HttpServlet {

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

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

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

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

User user = new User(username, password, email, "user");

UserDAO userDAO = new UserDAO();

boolean isRegistered = userDAO.registerUser(user);

if (isRegistered) {

response.sendRedirect("login.jsp");

} else {

request.setAttribute("error", "Registration failed.");

request.getRequestDispatcher("register.jsp").forward(request, response);

}

}

}

### ****7.4 TicketBookingServlet****

Books a seat and inserts a new row into the bookings table.

@WebServlet("/bookTicket")public class TicketBookingServlet extends HttpServlet {

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

int userId = ((User) request.getSession().getAttribute("user")).getId();

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

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

Booking booking = new Booking(userId, busId, seatNumber);

BookingDAO bookingDAO = new BookingDAO();

boolean isBooked = bookingDAO.bookTicket(booking);

if (isBooked) {

response.sendRedirect("home.jsp?success=booked");

} else {

response.sendRedirect("home.jsp?error=booking\_failed");

}

}

}

### ****7.5 AdminServlet (Optional but Useful)****

Can handle multiple admin operations: view users, modify buses, delete records.

@WebServlet("/admin")public class AdminServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

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

if ("viewUsers".equals(action)) {

List<User> users = new UserDAO().getAllUsers();

request.setAttribute("userList", users);

request.getRequestDispatcher("viewUsers.jsp").forward(request, response);

}

// Add more actions like 'deleteUser', 'addBus', etc.

}

}

### ****7.6 LogoutServlet****

Clears the session and logs the user out.

@WebServlet("/logout")public class LogoutServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

request.getSession().invalidate();

response.sendRedirect("login.jsp");

}

}

### ****7.7 Session Management Tips****

Always check session.getAttribute("user") before accessing protected pages.

Use filters if needed to protect routes like /admin/\*

Store minimal info in session: only user ID and role.

### ****7.8 Servlet Mapping Summary****

| **Servlet** | **URL Mapping** | **Description** |
| --- | --- | --- |
| LoginServlet | /login | Handles login |
| RegisterServlet | /register | Handles registration |
| TicketBookingServlet | /bookTicket | Handles ticket booking |
| AdminServlet | /admin | Admin operations |
| LogoutServlet | /logout | Clears session and redirects out |

****JSP Pages (Presentation Layer)****

## ****Chapter 8****

****JSP Pages (Presentation Layer)****

This chapter breaks down the core **JSP files** that handle front-end views, user interactions, and output rendering. These JSPs are where HTML meets Java using JSTL (JSP Standard Tag Library), Scriptlets, or JSTL expressions for dynamic content.

### ****8.1 login.jsp****

**Purpose:** Login form for users and admins.

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

<html>

<head><title>Login</title></head>

<body>

<h2>Login</h2>

<form method="post" action="login">

Username: <input type="text" name="username" required /><br>

Password: <input type="password" name="password" required /><br>

<input type="submit" value="Login" />

</form>

<c:if test="${not empty error}">

<p style="color:red">${error}</p>

</c:if>

<a href="register.jsp">New user? Register here</a>

</body>

</html>

### ****8.2 register.jsp****

**Purpose:** Registration form for new users.

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

<html>

<head><title>Register</title></head>

<body>

<h2>Register</h2>

<form method="post" action="register">

Username: <input type="text" name="username" required /><br>

Email: <input type="email" name="email" required /><br>

Password: <input type="password" name="password" required /><br>

<input type="submit" value="Register" />

</form>

<c:if test="${not empty error}">

<p style="color:red">${error}</p>

</c:if>

<a href="login.jsp">Already have an account? Login</a>

</body>

</html>

### ****8.3 home.jsp****

**Purpose:** Home page after login, displays buses and booking form.

<%@ page import="java.util.\*, models.Bus" %>

<%@ page session="true" %>

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

<jsp:useBean id="user" scope="session" class="models.User" />

<html>

<head><title>Home</title></head>

<body>

<h2>Welcome, ${user.username}!</h2>

<a href="logout">Logout</a>

<h3>Available Buses</h3>

<table border="1">

<tr><th>Name</th><th>From</th><th>To</th><th>Departure</th><th>Arrival</th><th>Seats</th><th>Book</th></tr>

<%

List<Bus> buses = (List<Bus>) request.getAttribute("busList");

for (Bus bus : buses) {

%>

<tr>

<td><%= bus.getBusName() %></td>

<td><%= bus.getSource() %></td>

<td><%= bus.getDestination() %></td>

<td><%= bus.getDepartureTime() %></td>

<td><%= bus.getArrivalTime() %></td>

<td><%= bus.getTotalSeats() %></td>

<td>

<form action="bookTicket" method="post">

<input type="hidden" name="busId" value="<%= bus.getId() %>"/>

Seat No: <input type="number" name="seatNumber" required min="1" max="<%= bus.getTotalSeats() %>"/>

<input type="submit" value="Book"/>

</form>

</td>

</tr>

<% } %>

</table>

</body>

</html>

### ****8.4 adminDashboard.jsp****

**Purpose:** Admin panel to view users/bookings/settings.

<html>

<head><title>Admin Dashboard</title></head>

<body>

<h2>Admin Dashboard</h2>

<a href="logout">Logout</a> |

<a href="admin?action=viewUsers">View Users</a> |

<a href="manageBookings.jsp">Manage Bookings</a> |

<a href="settings.jsp">Settings</a>

</body>

</html>

### ****8.5 viewUsers.jsp****

Displays all users for the admin.

<%@ page import="java.util.\*, models.User" %>

<%

List<User> users = (List<User>) request.getAttribute("userList");

%>

<html>

<head><title>View Users</title></head>

<body>

<h2>All Users</h2>

<table border="1">

<tr><th>Username</th><th>Email</th><th>Role</th></tr>

<%

for (User user : users) {

%>

<tr>

<td><%= user.getUsername() %></td>

<td><%= user.getEmail() %></td>

<td><%= user.getRole() %></td>

</tr>

<% } %>

</table>

<a href="adminDashboard.jsp">Back</a>

</body>

</html>

### ****8.6 manageBookings.jsp (Optional)****

Admin can view/cancel bookings.

### ****8.7 settings.jsp (Optional)****

Configuration and admin preferences.

### ****8.8 error.jsp / 404.jsp****

Generic fallback for errors or broken URLs.

jsp

<html>

<head><title>Error</title></head>

<body>

<h2>Error Occurred</h2>

<p>${errorMessage}</p>

<a href="home.jsp">Go Back</a>

</body>

</html>

## ****DAO Layer (Data Access Object)****

## ****Chapter 9****

****DAO Layer (Data Access Object)****

TDAO classes abstract away JDBC logic and provide methods to interact with the underlying database for each entity (User, Bus, Booking, etc.). This chapter covers major DAO implementations used in the project.

### ****9.1 UserDAO.java****

**Responsibilities:**

Register user

Login verification

Get user details

Admin: fetch all users

package dao;import java.sql.\*;import models.User;import utils.DBUtil;import java.util.\*;

public class UserDAO {

private Connection conn;

public UserDAO() {

conn = DBUtil.getConnection();

}

public boolean registerUser(User user) {

String sql = "INSERT INTO users(username, password, email, role) VALUES (?, ?, ?, ?)";

try (PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setString(1, user.getUsername());

ps.setString(2, user.getPassword());

ps.setString(3, user.getEmail());

ps.setString(4, user.getRole());

return ps.executeUpdate() > 0;

} catch (SQLException e) {

e.printStackTrace();

return false;

}

}

public User login(String username, String password) {

String sql = "SELECT \* FROM users WHERE username=? AND password=?";

try (PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setString(1, username);

ps.setString(2, password);

ResultSet rs = ps.executeQuery();

if (rs.next()) {

return new User(rs.getInt("id"), rs.getString("username"),

rs.getString("email"), rs.getString("role"));

}

} catch (SQLException e) {

e.printStackTrace();

}

return null;

}

public List<User> getAllUsers() {

List<User> list = new ArrayList<>();

String sql = "SELECT \* FROM users";

try (PreparedStatement ps = conn.prepareStatement(sql);

ResultSet rs = ps.executeQuery()) {

while (rs.next()) {

list.add(new User(rs.getInt("id"), rs.getString("username"),

rs.getString("email"), rs.getString("role")));

}

} catch (SQLException e) {

e.printStackTrace();

}

return list;

}

}

### ****9.2 BusDAO.java****

**Responsibilities:**

Add/update buses (optional)

Fetch all available buses

Get bus by ID

package dao;import java.sql.\*;import java.util.\*;import models.Bus;import utils.DBUtil;

public class BusDAO {

private Connection conn = DBUtil.getConnection();

public List<Bus> getAllBuses() {

List<Bus> list = new ArrayList<>();

String sql = "SELECT \* FROM buses";

try (PreparedStatement ps = conn.prepareStatement(sql);

ResultSet rs = ps.executeQuery()) {

while (rs.next()) {

list.add(new Bus(rs.getInt("id"), rs.getString("bus\_name"),

rs.getString("source"), rs.getString("destination"),

rs.getString("departure\_time"), rs.getString("arrival\_time"),

rs.getInt("total\_seats")));

}

} catch (SQLException e) {

e.printStackTrace();

}

return list;

}

public Bus getBusById(int busId) {

String sql = "SELECT \* FROM buses WHERE id=?";

try (PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setInt(1, busId);

ResultSet rs = ps.executeQuery();

if (rs.next()) {

return new Bus(rs.getInt("id"), rs.getString("bus\_name"),

rs.getString("source"), rs.getString("destination"),

rs.getString("departure\_time"), rs.getString("arrival\_time"),

rs.getInt("total\_seats"));

}

} catch (SQLException e) {

e.printStackTrace();

}

return null;

}

}

### ****9.3 BookingDAO.java**** (Optional if booking feature is implemented)

**Responsibilities:**

Create a booking

Fetch bookings by user/admin

Cancel booking

Join with payment or bus info (optional)

package dao;import java.sql.\*;import java.util.\*;import models.Booking;import utils.DBUtil;

public class BookingDAO {

private Connection conn = DBUtil.getConnection();

public boolean createBooking(Booking b) {

String sql = "INSERT INTO bookings(user\_id, bus\_id, booking\_time, seat\_number, status) VALUES (?, ?, ?, ?, 'booked')";

try (PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setInt(1, b.getUserId());

ps.setInt(2, b.getBusId());

ps.setString(3, b.getBookingTime());

ps.setInt(4, b.getSeatNumber());

return ps.executeUpdate() > 0;

} catch (SQLException e) {

e.printStackTrace();

return false;

}

}

public List<Booking> getBookingsByUser(int userId) {

List<Booking> list = new ArrayList<>();

String sql = "SELECT \* FROM bookings WHERE user\_id=?";

try (PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setInt(1, userId);

ResultSet rs = ps.executeQuery();

while (rs.next()) {

list.add(new Booking(rs.getInt("id"), rs.getInt("user\_id"),

rs.getInt("bus\_id"), rs.getString("booking\_time"),

rs.getInt("seat\_number"), rs.getString("status")));

}

} catch (SQLException e) {

e.printStackTrace();

}

return list;

}

public boolean cancelBooking(int bookingId) {

String sql = "UPDATE bookings SET status='cancelled' WHERE id=?";

try (PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setInt(1, bookingId);

return ps.executeUpdate() > 0;

} catch (SQLException e) {

e.printStackTrace();

return false;

}

}

}

### ****9.4 DBUtil.java****

A reusable JDBC utility to open DB connections.

package utils;import java.sql.\*;

public class DBUtil {

private static Connection conn;

public static Connection getConnection() {

if (conn == null) {

try {

Class.forName("org.sqlite.JDBC");

conn = DriverManager.getConnection("jdbc:sqlite:/path/to/your/bus.db");

} catch (Exception e) {

e.printStackTrace();

}

}

return conn;

}

}

## ****Models (JavaBeans / POJOs)****

## ****Chapter 10****

### ****Models (JavaBeans / POJOs)****

Each model class represents a table in your database (users, buses, bookings, payments). Let's break them down one by one.

### ****10.1 User.java****

package models;

public class User {

private int id;

private String username;

private String password;

private String email;

private String role;

// Constructor without ID (used during registration)

public User(String username, String password, String email, String role) {

this.username = username;

this.password = password;

this.email = email;

this.role = role;

}

// Constructor with ID (used when retrieving user from DB)

public User(int id, String username, String email, String role) {

this.id = id;

this.username = username;

this.email = email;

this.role = role;

}

// Getters and Setters

public int getId() { return id; }

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

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 getEmail() { return email; }

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

public String getRole() { return role; }

public void setRole(String role) { this.role = role; }

}

### ****10.2 Bus.java****

package models;

public class Bus {

private int id;

private String busName;

private String source;

private String destination;

private String departureTime;

private String arrivalTime;

private int totalSeats;

public Bus(int id, String busName, String source, String destination,

String departureTime, String arrivalTime, int totalSeats) {

this.id = id;

this.busName = busName;

this.source = source;

this.destination = destination;

this.departureTime = departureTime;

this.arrivalTime = arrivalTime;

this.totalSeats = totalSeats;

}

// Getters and Setters

public int getId() { return id; }

public String getBusName() { return busName; }

public String getSource() { return source; }

public String getDestination() { return destination; }

public String getDepartureTime() { return departureTime; }

public String getArrivalTime() { return arrivalTime; }

public int getTotalSeats() { return totalSeats; }

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

public void setBusName(String busName) { this.busName = busName; }

public void setSource(String source) { this.source = source; }

public void setDestination(String destination) { this.destination = destination; }

public void setDepartureTime(String departureTime) { this.departureTime = departureTime; }

public void setArrivalTime(String arrivalTime) { this.arrivalTime = arrivalTime; }

public void setTotalSeats(int totalSeats) { this.totalSeats = totalSeats; }

}

### ****10.3 Booking.java****

package models;

public class Booking {

private int id;

private int userId;

private int busId;

private String bookingTime;

private int seatNumber;

private String status;

public Booking(int id, int userId, int busId, String bookingTime, int seatNumber, String status) {

this.id = id;

this.userId = userId;

this.busId = busId;

this.bookingTime = bookingTime;

this.seatNumber = seatNumber;

this.status = status;

}

// Getters and Setters

public int getId() { return id; }

public int getUserId() { return userId; }

public int getBusId() { return busId; }

public String getBookingTime() { return bookingTime; }

public int getSeatNumber() { return seatNumber; }

public String getStatus() { return status; }

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

public void setUserId(int userId) { this.userId = userId; }

public void setBusId(int busId) { this.busId = busId; }

public void setBookingTime(String bookingTime) { this.bookingTime = bookingTime; }

public void setSeatNumber(int seatNumber) { this.seatNumber = seatNumber; }

public void setStatus(String status) { this.status = status; }

}

### ****10.4 Payment.java****

package models;

public class Payment {

private int id;

private int bookingId;

private double amount;

private String paymentStatus;

private String paymentTime;

public Payment(int id, int bookingId, double amount, String paymentStatus, String paymentTime) {

this.id = id;

this.bookingId = bookingId;

this.amount = amount;

this.paymentStatus = paymentStatus;

this.paymentTime = paymentTime;

}

// Getters and Setters

public int getId() { return id; }

public int getBookingId() { return bookingId; }

public double getAmount() { return amount; }

public String getPaymentStatus() { return paymentStatus; }

public String getPaymentTime() { return paymentTime; }

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

public void setBookingId(int bookingId) { this.bookingId = bookingId; }

public void setAmount(double amount) { this.amount = amount; }

public void setPaymentStatus(String paymentStatus) { this.paymentStatus = paymentStatus; }

public void setPaymentTime(String paymentTime) { this.paymentTime = paymentTime; }

}

Each of these model classes is lightweight, serializable (optionally), and used for transferring data between controller, DAO, and views.

**References**

**Chapter 11**

**References**

### ****11.1 – Books****

**Head First Servlets and JSP (2nd Edition)**  
By Bryan Basham, Kathy Sierra, Bert Bates  
– A comprehensive and beginner-friendly book on servlets and JSP, aligned with Java EE standards.

**Java: The Complete Reference (11th Edition)**  
By Herbert Schildt  
– Covers core Java, OOP principles, and in-depth JDBC for database connectivity.

**Core Servlets and JavaServer Pages Volume 1**  
By Marty Hall and Larry Brown  
– Ideal for understanding servlet architecture, deployment, and integration with databases.

**Beginning JSP, JSF and Tomcat: Java Web Development**  
By Giulio Zambon  
– A practical guide for building complete Java-based web apps using JSP and Servlets.

**Effective Java (3rd Edition)**  
By Joshua Bloch  
– Focuses on Java best practices, ideal for writing robust and maintainable code.

**Murach’s Java Servlets and JSP (3rd Edition)**  
By Joel Murach  
– Step-by-step examples for real-world projects involving MVC, database, and UI integration.

**Database Systems: The Complete Book**  
By Hector Garcia-Molina, Jeffrey Ullman, Jennifer Widom  
– In-depth understanding of database concepts, schema design, and SQL.

**Learning SQL (2nd Edition)**  
By Alan Beaulieu  
– Great for mastering SQL queries, relational database design, and practical use cases.

**Clean Code: A Handbook of Agile Software Craftsmanship**  
By Robert C. Martin  
– Teaches coding ethics, structure, and refactoring principles—very useful for backend logic.

**Professional Java for Web Applications**  
By Nicholas S. Williams  
– Covers advanced Java EE concepts, including servlet filters, listeners, JSP custom tags, and REST API development.

### ****11.2 – Websites****

**Oracle Java Official Documentation**  
[https://docs.oracle.com/javase/8/docs/](https://docs.oracle.com/javase/8/docs/" \t "/home/akali/Documents\\x/_new)  
– The go-to source for all Java-related syntax, class documentation, and examples.

**Jakarta EE / Servlet Specification**  
https://jakarta.ee/specifications/servlet/  
– Reference for Servlet API lifecycle and features.

**W3Schools – Java Servlets & JSP**  
https://www.w3schools.com/java/java\_servlets.asp  
– Simple tutorials for beginners learning JSP and servlet development.

**GeeksForGeeks – Java Web Programming**  
https://www.geeksforgeeks.org/java-servlets/  
– Practical examples, common issues, and student-friendly guides.

**Stack Overflow**  
[https://stackoverflow.com/](https://stackoverflow.com/" \t "/home/akali/Documents\\x/_new)  
– Real-world Q&A community for solving development bugs and configuration errors.