**Multi-client Service Website**

**Submitted By:**

**NAME : SRI HARSHANALI.V**

**ROLL NO: BU21CSEN0300418**

**PROBLEM STATEMENT:**

The need for a centralized platform where service providers can offer their services to customers has become increasingly important. The main challenges include:

* **User Management**: Managing multiple users with different roles (providers, customers, admin).
* **Service Booking**: Allowing customers to easily browse and book services.
* **Payment Processing**: Ensuring secure transactions between users.
* **Review System**: Enabling feedback for service quality assurance.
* **Scalability**: Designing a system that can grow with increased users and services.

**SOLUTION :**

The solution involves a full-stack web application structured as follows:

* **Frontend**: ReactJS or AngularJS for a dynamic user interface.
* **Backend**: NodeJS with Express.js or Spring Boot for API services.
* **Database**: MongoDB or MySQL for data storage.

**Key Components**

**1. User Authentication and Management**

* **User Registration/Login**:
  + Implement secure user registration using bcrypt for password hashing.
  + Use JWT for session management.
* **Role-based Access Control**:
  + Differentiate functionalities based on user roles (service provider, customer, admin).

**2. Service Listings**

* **CRUD Operations**:
  + Allow service providers to create, read, update, and delete their service listings.
* **Search and Filter**:
  + Implement search functionality to help users find services by category, location, or ratings.

**3. Booking System**

* **Availability Calendar**:
  + Service providers can manage their availability using a calendar interface.
* **Booking Confirmation**:
  + Implement a booking confirmation system that sends notifications via email or SMS.

**4. Payment Integration**

* **Payment Gateway**:
  + Use Stripe or PayPal API for secure payment processing.
* **Transaction Management**:
  + Record transaction details in the database, ensuring users can view their payment history.

**5. Review and Rating System**

* **User Feedback**:
  + Allow customers to rate services and leave reviews after the service is completed.
* **Moderation Tools**:
  + Admins can monitor and moderate reviews to prevent abuse.

**6. Admin Dashboard**

* **User Management**:
  + Admins can view, edit, or deactivate user accounts.
* **Service Oversight**:
  + Admins can manage service listings and resolve disputes.

**3. Technology Stack**

* **Frontend**:
  + HTML, CSS, Bootstrap, ReactJS (or AngularJS).
* **Backend**:
  + NodeJS with Express.js (or Java with Spring Boot).
* **Database**:
  + MongoDB (for flexible schemas) or MySQL (for structured data).

**Implementation Steps:**

**Frontend Development**

1. **Setup Project Structure**
   * Create a basic structure for your HTML, CSS, and JavaScript files.
2. **User Interface Design**
   * Use HTML and Bootstrap to create responsive pages:
     + **Home Page**: Introduction and service overview.
     + **Registration/Login Page**: Forms for user authentication.
     + **Profile Page**: Service provider profile with service listings.
     + **Booking Page**: Interface for booking services.
     + **Admin Dashboard**: Overview of users, services, and reviews.
3. **Client-Side Logic**
   * Use JavaScript to handle form submissions and dynamic content updates (like service filtering).

**Backend Development**

1. **Set Up Spring Boot Project**
   * Use Spring Initializr to create a new project with dependencies for Spring Web, Spring Data JPA, and MySQL Driver.
2. **Database Configuration**
   * Configure MySQL connection in application.properties

**CODE:**

spring.datasource.url=jdbc:mysql://localhost:3306/your\_database

spring.datasource.username=your\_username

spring.datasource.password=your\_password

spring.jpa.hibernate.ddl-auto=update

**Create Entity Classes**

* Define JPA entity classes for users, services, bookings, and reviews.
* **CODE:**

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String username;

private String password; // Should be hashed

private String role; // "provider" or "customer"

// Getters and setters

}

1. **Repository Interfaces**
   * Create repository interfaces for database interactions:
   * **CODE:**

public interface UserRepository extends JpaRepository<User, Long> {

Optional<User> findByUsername(String username);

}

1. **Service Layer**
   * Implement service classes to handle business logic (e.g., user registration, service creation, bookings).
2. **Controller Layer**
   * Create REST controllers to handle API requests:
   * **CODE:**

@RestController

@RequestMapping("/api/users")

public class UserController {

@Autowired

private UserService userService;

@PostMapping("/register")

public ResponseEntity<User> registerUser(@RequestBody User user) {

// Logic for user registration

}

}

**Payment Processing**

1. **Integrate Payment Gateway**
   * Choose a payment provider (like Stripe) and follow their API documentation to set up secure payment processing.
   * Create a payment controller to handle payment requests.

**Testing**

* Perform unit tests for the backend services and integration tests for the API endpoints.
* Use manual testing to ensure the frontend works correctly and user interactions are smooth.

**Deployment**

1. **Build the Application**
   * Use Maven or Gradle to build the Spring Boot application.
2. **Deploy on a Server**
   * Deploy the application on a cloud provider (like AWS, Heroku) and configure the database connection.
3. **Domain and SSL**
   * Set up a domain name and enable SSL for secure connections.

**4. Security Considerations**

* **Data Encryption**: Use HTTPS to secure data in transit.
* **Input Validation**: Sanitize inputs to prevent SQL injection and XSS attacks.
* **Password Security**: Hash passwords before storing them using bcrypt.
* **Access Control**: Implement role-based access control to restrict access based on user roles.

**5. Scalability Considerations**

* **Database Optimization**: Use indexing and optimize queries for better performance.
* **Load Balancing**: Implement load balancers to handle increased traffic as the user base grows.
* **Microservices Architecture**: Consider transitioning to microservices for scalability if the application grows significantly.

Creating a complete output for a Multi-Client Service Website involves various components, including HTML pages, backend APIs, and database interactions. Below, I’ll provide sample outputs for key functionalities within this project.

**1. User Registration Output**

**Frontend (HTML Form):**

<!-- registration.html -->

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Register</title>

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">

</head>

<body>

<div class="container">

<h2>Register</h2>

<form id="registrationForm">

<div class="form-group">

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

<input type="text" class="form-control" id="username" required>

</div>

<div class="form-group">

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

<input type="password" class="form-control" id="password" required>

</div>

<button type="submit" class="btn btn-primary">Register</button>

</form>

</div>

<script>

document.getElementById('registrationForm').addEventListener('submit', function(event) {

event.preventDefault();

const username = document.getElementById('username').value;

const password = document.getElementById('password').value;

fetch('/api/users/register', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ username, password })

})

.then(response => response.json())

.then(data => {

alert('User registered successfully: ' + JSON.stringify(data));

})

.catch(error => {

console.error('Error:', error);

});

});

</script>

</body>

</html>

**Backend (Java Controller)**

@RestController

@RequestMapping("/api/users")

public class UserController {

@Autowired

private UserService userService;

@PostMapping("/register")

public ResponseEntity<User> registerUser(@RequestBody User user) {

User savedUser = userService.registerUser(user);

return ResponseEntity.status(HttpStatus.CREATED).body(savedUser);

}

}

**Sample Output after Registration**

{

"id": 1,

"username": "john\_doe",

"role": "customer"

}

**2. Service Listing Output**

**Frontend (HTML Page)**

<!-- services.html -->

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Available Services</title>

</head>

<body>

<h2>Available Services</h2>

<div id="servicesList"></div>

<script>

fetch('/api/services')

.then(response => response.json())

.then(services => {

const servicesList = document.getElementById('servicesList');

services.forEach(service => {

const serviceItem = document.createElement('div');

serviceItem.innerHTML = `<h4>${service.name}</h4><p>${service.description}</p>`;

servicesList.appendChild(serviceItem);

});

});

</script>

</body>

</html>

**Backend (Service Controller)**

@RestController

@RequestMapping("/api/services")

public class ServiceController {

@Autowired

private ServiceService serviceService;

@GetMapping

public ResponseEntity<List<Service>> getAllServices() {

List<Service> services = serviceService.findAllServices();

return ResponseEntity.ok(services);

}

}

**Sample Output for Services**

[

{

"id": 1,

"name": "House Cleaning",

"description": "Professional cleaning service for your home."

},

{

"id": 2,

"name": "Personal Training",

"description": "One-on-one personal training sessions."

}

]

**3. Booking Service Output**

**Frontend (HTML Booking Page)**

<!-- booking.html -->

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Book Service</title>

</head>

<body>

<h2>Book Service</h2>

<form id="bookingForm">

<label for="serviceId">Service ID:</label>

<input type="text" id="serviceId" required>

<label for="customerId">Customer ID:</label>

<input type="text" id="customerId" required>

<button type="submit">Book Service</button>

</form>

<script>

document.getElementById('bookingForm').addEventListener('submit', function(event) {

event.preventDefault();

const serviceId = document.getElementById('serviceId').value;

const customerId = document.getElementById('customerId').value;

fetch('/api/bookings', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ serviceId, customerId })

})

.then(response => response.json())

.then(data => {

alert('Booking successful: ' + JSON.stringify(data));

})

.catch(error => {

console.error('Error:', error);

});

});

</script>

</body>

</html>

**Backend (Booking Controller)**

@RestController

@RequestMapping("/api/bookings")

public class BookingController {

@Autowired

private BookingService bookingService;

@PostMapping

public ResponseEntity<Booking> bookService(@RequestBody Booking booking) {

Booking newBooking = bookingService.createBooking(booking);

return ResponseEntity.status(HttpStatus.CREATED).body(newBooking);

}

}

**Sample Output for Booking**

{

"id": 1,

"serviceId": 1,

"customerId": 1,

"status": "confirmed"

}

**4. Review Submission Output**

**Frontend (HTML Review Form)**

<!-- review.html -->

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Submit Review</title>

</head>

<body>

<h2>Submit Review</h2>

<form id="reviewForm">

<label for="serviceId">Service ID:</label>

<input type="text" id="serviceId" required>

<label for="rating">Rating (1-5):</label>

<input type="number" id="rating" min="1" max="5" required>

<label for="comment">Comment:</label>

<textarea id="comment" required></textarea>

<button type="submit">Submit Review</button>

</form>

<script>

document.getElementById('reviewForm').addEventListener('submit', function(event) {

event.preventDefault();

const serviceId = document.getElementById('serviceId').value;

const rating = document.getElementById('rating').value;

const comment = document.getElementById('comment').value;

fetch('/api/reviews', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ serviceId, rating, comment })

})

.then(response => response.json())

.then(data => {

alert('Review submitted: ' + JSON.stringify(data));

})

.catch(error => {

console.error('Error:', error);

});

});

</script>

</body>

</html>

**Backend (Review Controller)**

@RestController

@RequestMapping("/api/reviews")

public class ReviewController {

@Autowired

private ReviewService reviewService;

@PostMapping

public ResponseEntity<Review> submitReview(@RequestBody Review review) {

Review newReview = reviewService.submitReview(review);

return ResponseEntity.status(HttpStatus.CREATED).body(newReview);

}

}

**Sample Output for Review Submission**

{

"id": 1,

"serviceId": 1,

"rating": 5,

"comment": "Excellent service!"

}

**Conclusion**

This output showcases basic interactions for user registration, service listing, service booking, and review submission within a Multi-Client Service Website. You can expand these components with additional functionalities, error handling, and front-end enhancements as needed.