|  |
| --- |
| Close-up image showing the leaf-sides of two oversized books side-by-side on a bookshelf, with additional books in soft focus background |
| **THE DJ BOOKING SYSTEM PRESENTATION REPORT**  **Boitumelo Xaba: 221097570 | Indiphile Wopela: 222841176 | Alvaro Ferraz: 220075018 | Maphelo Shaun Tshapile: 213152231** |
|  |
|  |

**Table of Contents**

[**TERM 1 — INTRODUCTION (BACKGROUND AND ENTITIES)** 3](#_Toc211800275)

[**Background** 3](#_Toc211800276)

[**Key Entities And Their Relationships** 3](#_Toc211800277)

[**How The System Works** 3](#_Toc211800278)

[**TERM 2 — BACK-END (STRUCTURE, LANGUAGE, AND DATA)** 4](#_Toc211800279)

[**Application Layers And Responsibilities:** 4](#_Toc211800280)

[**TERM 3 - FRONT-END (TOOLS, STRUCTURE, AND COMMUNICATION)** 5](#_Toc211800281)

[**How Front-End And Back-End Communicate:** 5](#_Toc211800282)

[**TERM 4 — SECURITY, DESIGN PATTERNS, AND DESIGN PRINCIPLES** 6](#_Toc211800283)

[**THE CONCLUSION** 6](#_Toc211800284)

# **TERM 1 — INTRODUCTION (BACKGROUND AND ENTITIES)**

## **Background**

The DJ Booking System is a web application that helps event organizers find and hire DJs, manage events, reserve equipment, track payments, and gather reviews. It supports booking lifecycle from request to payment and post-event feedback.

## **Key Entities And Their Relationships**

DJ: a performer who can be booked. A DJ can have many Bookings and can be assigned to many Gigs.

Gig: an event that needs a DJ. A Gig may require Equipment and occupies a Timeslot. A Gig is usually linked to one DJ.

Booking: the record that ties a client to a Gig and a DJ for a date/time. Bookings create Payment records and may have Reviews.

Equipment: gear such as speakers and lights. Equipment can be attached to multiple Gigs.

Payment: a record of a client’s payment for a Booking. Each Payment links to a single Booking.

Review: feedback left by the client for a Booking (and indirectly for the DJ).

Admin: the user role that manages DJs, Gigs, Equipment, Bookings, and Payments.

Timeslot: the specific start and end time for a Gig or Booking.

## **How The System Works**

A client requests a Booking for a selected Gig and Timeslot. The system checks the DJ’s availability and equipment needs. If the booking is approved, the client pays, and a Payment record is created. After the event, the client can add a Review. Admins handle management tasks and resolve conflicts.

# **TERM 2 — BACK-END (STRUCTURE, LANGUAGE, AND DATA)**

Language and build: Java, built with Maven (“pom.xml” present).

Frameworks: Spring-style architecture (controllers, services, repositories) with configuration classes (for example “WebConfig.java”) and application properties.

Database: H2 is included for development and testing (files show “application-h2. properties”).

## **Application Layers And Responsibilities:**

Domain layer - Contains POJOs for the main entities: “DJ”, “Gig”, “Booking”, “Equipment”, “Payment”, “Review”, and “Admin”. These classes model the data stored in the database.

Factory layer - Factory classes (e.g., “DJFactory”, “BookingFactory”) create domain objects consistently and validate required fields when building objects.

Repository layer - Repository interfaces (e.g., “BookingRepository”, “DJRepository”) handle data access. They abstract database operations, so services don't deal with SQL directly.

Service layer - Service interfaces (“IBookingService”, “IDjService”, etc.) and implementations (“BookingService”, “DjService”, etc.) contain business rules: checking availability, validating booking rules, coordinating multi-step operations, and calling repositories.

Controller layer - Controllers (e.g., “BookingController”, “DjController”) expose HTTP endpoints to the front end. They accept requests, call services, and return responses (JSON or views).

Configuration and utilities - “WebConfig” and properties files configure web behaviour and database connections. Utilities provide helpers used across layers.

Database and persistence - The app uses relational tables mapped from domain classes (likely via JPA). H2 is configured for development and tests, with a properties file for connection settings. Repositories persist and retrieve entities, and services manage transactions.

How the back-end flows - Front end calls a REST endpoint. Controller receives the call, converts payload into domain objects (often using factories), calls the service to run business logic, and the service uses repositories to save or query the database. The controller returns success or error to the client.

# **TERM 3 - FRONT-END (TOOLS, STRUCTURE, AND COMMUNICATION)**

Framework and language: React single-page application (SPA) written in JavaScript (JSX).

Project type: Standard React app (looks like Create React App scaffolding with “index.js”, “App.js”, “reportWebVitals.js”). Build tools: Node/npm, defined in “package.json”.

Key parts of the front end - Entry and layout - “public/index.html” and “src/index.js” bootstrap the app. “src/App.js” holds the top-level component and routing logic. Components and pages “components/” contains pages and UI parts: “BookingForm”, “Dashboard”, “ManageDJsPage”, “ManageGigsPage”, “EquipmentPage”, “LoginPage”, “RegisterPage”, “PaymentPage”, “ReviewPage”, “AboutUs”, “ContactUs”. Each component has a CSS file for styling.

API layer - “src/api.js” centralizes HTTP calls to the back-end API. It likely sets base URL and handles common headers (e.g., authentication tokens). Tests and tooling. “App.test.js” and “setupTests.js” indicate unit testing with Jest and React Testing Library. “reportWebVitals.js” is present for performance metrics.

## **How Front-End And Back-End Communicate:**

The front end sends HTTP requests (GET, POST, PUT, DELETE) through “api.js” to back-end endpoints like “/api/bookings” or “/api/djs”. Responses are displayed, and local state updates reflect results.

# **TERM 4 — SECURITY, DESIGN PATTERNS, AND DESIGN PRINCIPLES**

- Authentication: The app has login and register pages. The back end should protect endpoints using a security layer (Spring Security is recommended). Tokens or sessions should be used securely (prefer httpOnly cookies).

Passwords and sensitive data: Use strong hashing (BCrypt) for passwords. Avoid storing raw payment details; integrate with a payment processor if possible. Input validation and sanitization: Validate on server and client. Use factories and service checks to reject bad data. Transport and secrets: Use HTTPS in production and store database credentials and secrets in environment variables or a secret manager. Access control: Restrict admin routes to admin users and protect all write operations.

Design patterns observed - Layered architecture: Controllers, Services, Repositories separate responsibilities. Factory pattern: Factories build domain objects and handle validation. Repository pattern: Repositories abstract data storage. Dependency injection: Spring injects service and repository dependencies, enabling easier testing. Interface-driven services: Service interfaces (“I\*” types) support loose coupling and testability.

Design principles - Single Responsibility: Classes do one job (controllers handle requests, services handle logic, repositories handle DB). Separation of Concerns: Front end handles UI; back-end handles data and rules. Open/Closed and Testability: Services and repositories are designed behind interfaces to allow changes without modifying callers and to make unit testing easy. DRY and reuse: Central “api.js” and factories reduce repeated code.

# **THE CONCLUSION**

The DJ Booking System is a well-structured web app with a React front end and a Java back end using Spring-style layers. The architecture maps closely to the UML: DJS, Gigs, Bookings, Equipment, Payments, Reviews, and Admins are modelled, stored, and managed through clear layers. With a few security and deployment enhancements, the system is ready for safe production use.