

**UE22CS352B - Object Oriented Analysis & Design**

Mini Project Report

Title

Restaurant booking system

***Submitted by:***

***Name : SRN (Team Members)***

|  |  |
| --- | --- |
| **NARU MEGHANA**  **NEHA GIRISH**  **NIDA FATHIMA NIKHITA G** | **PES2UG22CS341**  **PES2UG22CS346**  **PES2UG22CS353**  **PES2UG22CS359** |

Semester:6th Section:F

# Facultly Name

# Divya Ebenezer

**January - May 2025**

##### DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

FACULTY OF ENGINEERING

##### PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013) 100ft Ring Road, Bengaluru – 560 085, Karnataka, India

##### Problem Statement:

##### In many restaurants, table reservations are still handled manually via phone calls or on-site requests. This often leads to scheduling conflicts, overbookings, or underutilization of seating capacity. There is a lack of an efficient system to manage real-time table availability, especially during peak hours.

##### To address this issue, the objective is to develop an online restaurant booking system that allows customers to reserve tables by checking real-time availability based on date and time. The system must ensure that no double bookings occur by validating overlapping time slots before confirming a reservation. The platform should also support persistent data storage, enabling restaurant administrators to track, manage, and review upcoming reservations.

**Key Features of the Restaurant Booking System**

**1. Real-Time Table Availability Check**

* Checks if a selected table is available for the requested date and time.
* Prevents double bookings using smart time-overlap detection logic.

**2. Time Slot Conflict Detection**

* Ensures reservations don’t clash by validating arrival and departure times.
* Avoids scheduling conflicts to maintain a smooth booking process.

**3. Table Selection Based on Group Size**

* Tables are organized by seating capacity (e.g., 2, 4, 6, 8, 10).
* Users can choose appropriately sized tables for their party.

**4. Persistent Reservation Storage (MongoDB)**

* Bookings are saved in a database for future retrieval and management.
* Ensures data consistency across sessions and restarts.

**5. User-Friendly Frontend Interface**

* HTML-based pages for customers and admins to book or view reservations.
* Simple and intuitive navigation for easy access and use.

## Models:

Use Case Diagram:

**Class Diagram:**

# State Diagram:

**Activity Diagrams:**

1. Major Usecase
2. Minor Use case

Architecture Patterns, Design Principles, and Design Patterns:

# Architecture Patterns

# Model – View – Controller Pattern (MVC)

# 1.Model – backend/src/main/java/com/restaurant/booking/model

# Responsible for representing data and business logic.

# Classes:

# Reservation.java – Represents a reservation made by a customer.

# Table.java – Represents a table in the restaurant (ID, size).

# ReservationDocument.java – MongoDB-specific representation of a reservation document for persistence.

# Responsibility:

# Encapsulate and manage data.

# Provide structures passed between controller and service.

# Interact with the repository for database operations.

# 2. View – frontend/\*.html

# Responsible for presenting data to the user (UI).

# Files:

# index.html – Main user-facing interface for table booking.

# admin.html – Admin panel for viewing reservations (possibly).

# login.html – Login page for customer/admin authentication.

# Responsibility:

# Provide user input forms for booking and login.

# Display reservation feedback or table availability.

# Communicate with the controller layer (via forms or JS calls to backend endpoints).

# 3. Controller – backend/src/main/java/com/restaurant/booking/controller

# Handles user requests, processes them, and returns responses.

# Class:

# BookingController.java – Acts as the entry point for HTTP requests related to booking.

# Responsibility:

# Map user actions (HTTP requests) to the appropriate service methods.

# Pass data between the view (frontend) and service (business logic).

# Return responses (HTML, JSON, etc.).

**4. Service Layer – backend/src/main/java/com/restaurant/booking/service**

Encapsulates business logic and coordination.

Class:

* BookingService.java – Core logic for handling reservations, checking availability, saving to DB.

Responsibility:

* Validate and process reservation requests.
* Check for table conflicts.
* Communicate with the repository layer for CRUD operations.

**5. Repository Layer – backend/src/main/java/com/restaurant/booking/repository**

Handles persistence and data access.

Class:

* ReservationRepository.java – Interface extending MongoRepository to interact with MongoDB.

Responsibility:

* Abstract database operations (save, findAll, etc.).
* Keep business logic decoupled from database handling.

# Design Principles

# 1.Single Responsibility Principle (SRP)

# Each class or module should have one and only one reason to change.

# Application:

# BookingService handles only reservation logic.

# ReservationRepository manages database operations.

# Models (Reservation, Table, ReservationDocument) represent structured data only.

# Benefit: Clear separation of concerns, making the system easier to test and maintain.

# 2.Open/Closed Principle (OCP)

# Software components should be open for extension but closed for modification.

# Application:

# The system can support additional features (e.g., cancellation, dynamic table allocation algorithms) by extending services or creating new strategies without altering core logic.

# Overlapping time logic can be extended with the Strategy Pattern.

# Benefit: Enables future enhancements without breaking existing functionality.

# 3. Liskov Substitution Principle (LSP)

# Objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

# Application:

# Repository abstraction (MongoRepository) can be replaced with another database implementation if needed.

# Benefit: Promotes interchangeable components and system flexibility.

# 4.Interface Segregation Principle (ISP)

# Clients should not be forced to depend on interfaces they don’t use.

# Application:

# The system uses focused interfaces (ReservationRepository) that provide only required CRUD operations.

# Benefit: Reduces coupling and keeps the codebase clean and minimal.

# 5.Dependency Inversion Principle (DIP)

# High-level modules should not depend on low-level modules. Both should depend on abstractions.

# Application:

# BookingService depends on the abstract ReservationRepository interface, not on its concrete implementation.

# Dependency Injection is used via Spring's @Autowired.

# Benefit: Enhances testability, flexibility, and code reuse.

# Design Patterns

Below are four design patterns that are implemented or can be integrated into the restaurant booking system. Each pattern has been applied in a specific area of the project, and comments have been included to clarify the implementation:

1. Singleton Pattern (Creational)

Where in the Project:

* The Singleton pattern is implemented in the BookingService class by annotating it with @Service. This ensures that Spring manages the service as a singleton, meaning only one instance of the BookingService is created and used throughout the application.

Why It Fits:

* The Singleton pattern ensures that there is a single shared state across the entire application, maintaining consistency in the service's behavior.

\

### Github link to the Codebase:

<https://github.com/neeharika-anand/grocery-store>

# Screenshots

## UI:

Individual contributions of the team members:

|  |  |
| --- | --- |
| **Name** | **Module worked on** |
|  |  |
|  |  |
|  |  |
|  |  |