**<EventEase >**

Project documentation

Student(s): Gîrda Nora- Marcela

**Group: 30434**

**Contents**

[I Project specification](#_heading=h.gjdgxs) **3**

[1.1 Domain Model Diagram](#_heading=h.30j0zll) 3

[II Use-Case model](#_heading=h.1fob9te) **3**

[2.1 Users and stakeholders](#_heading=h.f2irg1azp7rm) 4

[2.2 Use-Case identification](#_heading=h.3znysh7) 4

[2.3 UML Use-Case diagram](#_heading=h.2et92p0) 4

[III Architectural design](#_heading=h.tyjcwt) **5**

[3.1 Conceptual architecture](#_heading=h.3dy6vkm) 5

[3.2 Package diagram](#_heading=h.1t3h5sf) 5

[3.3 Class diagram](#_heading=h.4d34og8) 5

[3.4 Database (E-R/Data model) diagram](#_heading=h.2s8eyo1) 6

[3.5 Sequence diagram](#_heading=h.17dp8vu) 6

[3.6 Activity diagram](#_heading=h.3rdcrjn) 6

[IV Supplementary specifications](#_heading=h.26in1rg) **6**

[4.1 Non-functional requirements](#_heading=h.lnxbz9) 6

[4.2 Design constraints](#_heading=h.35nkun2) 6

[V Testing](#_heading=h.1ksv4uv) **7**

[5.1 Testing methods/frameworks](#_heading=h.44sinio) 7

[5.2 Future improvements](#_heading=h.2jxsxqh) 7

[VI Bibliography](#_heading=h.z337ya) **7**

# I Project specification

The project represents a client-server web application designed to provide a platform for event hall owners to efficiently manage venues, bookings, and event details. Built using Java Spring for the backend and React for the frontend, the application aims to offer an intuitive and structured way to plan, customize, and organize events while ensuring a seamless experience for both administrators and clients.

The system allows:

* Users to register/login into the platform with role-based access (admin and client).
* Admins to manage event venues, handling multiple event locations and their availability.
* Clients to book venues, customize event details (guest count, menu, decorations, etc.), and make secure payments.
* Admins to oversee and modify event details, send notifications, and approve last-minute changes requested by clients.
* Clients to visualize venue layouts, arrange tables interactively, and plan seating arrangements.
* A built-in calendar system to track venue availability and prevent double bookings.
* Secure payment processing via an integrated payment gateway, ensuring a hassle-free transaction experience.

Additionally, the system may include notifications, reminders, and other features features to enhance the efficiency of event planning. The structured role-based system ensures that admins have full control over event management, while clients can seamlessly plan their special occasions within a user-friendly, well-organized interface.

## 1.1 Domain Model Diagram

A black background with white rectangles

AI-generated content may be incorrect.

# II Use-Case model

The Use-Case Model for the Events Managing App is designed to define user interactions by structuring functionalities around distinct roles: Admin and Client. Each role has specific permissions, ensuring controlled access to features such as event management, venue selection, booking, and customizations.

## 2.1 Users and stakeholders

**Users:**

The application will have two primary types of users, each with different interactions based on their permissions:

* **Admin:**
  + Manages multiple event venues.
  + Creates, updates, and deletes events.
  + Manages event settings, pricing, and decorations.
  + Has full CRUD access to venue and event data.
  + Can approve or deny client changes after a deadline.
  + Can send notifications to clients.
* **Client:**
  + Registers and books event venues.
  + Chooses event details (guest count, menu, drinks, decorations, music, etc.).
  + Can modify event details before a set deadline.
  + Arranges seating plans using an interactive venue layout.
  + Makes secure payments for bookings.
  + Receives notifications regarding booking status and updates.

**Stakeholders:**

* **Event Hall Owners:** The primary beneficiaries who use the system to manage event planning and make sure that the events run smoothly so that the business is going to be good.
* **Clients (Event Organizers):** Individuals booking venues for weddings, baptisms, or celebrations, who want the best services for their events.
* **Service Providers:** These once can be the music, decorations, sweets providers who provide additional services to what the venue has to offer.

## 2.2 Use-Case identification

**Use Case Name: Register/Login**

* **Level:** User-Goal
* **Main Actor:** Admin, Client
* **Main Success Scenario:**
  1. The user navigates to the home page.
  2. The user selects **Register** or **Login**.
  3. The system displays a form for entering credentials.
  4. The user completes the form and submits it.
  5. The system validates the data.
  6. If valid, the user is granted access and redirected to their dashboard.
* **Extension (Negative Scenario):**
  1. If the user enters an **invalid email or password**, an error message is displayed.
  2. If the user tries to register with an **already existing email**, an error is displayed.

**Use Case Name: Venue Booking**

* **Level:** User-Goal
* **Main Actor:** Client
* **Main Success Scenario:**
  1. The client logs into the system.
  2. The client navigates to the **venue selection** page.
  3. The system displays available venues and their details.
  4. The client selects a venue and event type.
  5. The system prompts the client to confirm details and pay a deposit.
  6. Upon successful payment, the booking is **confirmed** and linked to the client’s account.
* **Extension (Negative Scenario):**
  1. If the venue is already booked on the selected date, an **error message** is displayed.
  2. If the payment fails, the system prompts the client to **retry or choose another method**.

**Use Case Name: Modify Event Details**

* **Level:** User-Goal
* **Main Actor:** Client, Admin
* **Main Success Scenario:**
  1. The client selects a **confirmed booking**.
  2. The client updates details (guest count, menu, decorations, etc.).
  3. If modifications are **before the deadline**, changes are saved automatically.
  4. If modifications are **after the deadline**, an **admin approval request** is sent.
  5. The admin approves/rejects the modifications.
  6. If approved, the changes are updated, and the client is notified.
* **Extension (Negative Scenario):**
  1. If the event is less than **2 weeks away**, modifications require **admin approval**.
  2. If the admin **rejects the request**, the client receives a **notification with the reason**.

**Use Case Name: Manage Payments**

* **Level:** User-Goal
* **Main Actor:** Client
* **Main Success Scenario:**
  1. The client confirms an event booking.
  2. The system displays the **total price** based on the chosen options.
  3. The client selects a **payment method** and proceeds with the payment.
  4. The payment gateway securely processes the transaction.
  5. Upon successful payment, a **confirmation message** is displayed, and the booking is finalized.
* **Extension (Negative Scenario):**
  1. If the payment **fails**, the system prompts the client to **retry or select another method**.
  2. If the **server is down**, the transaction is retried later.

**Use Case Name: Send Notifications**

* **Level:** Summary
* **Main Actor:** Admin
* **Main Success Scenario:**
  1. The admin logs into the system.
  2. The admin navigates to the **Notifications** page.
  3. The system displays **pending notifications**.
  4. The admin selects a notification type (booking confirmation, deadline warning, etc.).
  5. The notification is sent to the respective **clients**.
* **Extension (Negative Scenario):**
  1. If the **client's email is invalid**, the system logs a **failed notification attempt**.

## 2.3 UML Use-Case diagram

A screen shot of a computer

AI-generated content may be incorrect.

# III Architectural design

The application uses a **Layered architectural style** to help organize the system into logical sections and group related classes together. This approach fits my application well because it allows me to **separate responsibilities** (like the UI, business logic, and database), making the project easier to **read, manage, and debug**. If something goes wrong, it becomes clearer **which layer the issue comes from**, allowing for faster troubleshooting and better structure during development.

## 3.1 Conceptual architecture

The application will be a **web application**, following a typical **Client-Server** pattern, as its main purpose is to make event management easy and approachable. The client will be the user who interact with the frontend, while the server will be the backend application, which handles business logic, authentication, and database access, communicating via **HTTP** requests.

The **Events Managing App** follows a **5-Layer Architecture** that ensures high modularity, maintainability, and a clean separation of concerns. This layered approach is ideal for web applications that handle multiple user roles (admin and client), complex business rules, and data-intensive operations. Each layer has a well-defined responsibility and interacts with adjacent layers to ensure a smooth flow of data and logic.

#### ****Presentation Layer (Frontend)****

* Built using **React**.
* Responsible for the **user interface** and handling interactions from both clients and admins.
* Sends requests and receives data via **RESTful APIs** exposed by the backend.
* Displays forms, calendars, event configuration tools, and seating arrangement interfaces.

#### ****Controller Layer****

* Built using **Spring Boot Controllers**.
* Acts as the **entry point** for all client and admin requests.
* **Maps API routes** to specific services and handles routing based on user role (admin/client).
* Validates input at a high level and delegates detailed processing to the service (business) layer.

**Business Layer (Service Layer)**

* Contains the core application logic, such as:
  + Validating guest numbers, event rules, and payment deadlines.
  + Enforcing restrictions (e.g., changes within 2 weeks need admin approval).
  + Sending notifications to clients.
* Manages workflow logic, error handling, and interaction coordination between controller and persistence layers.

**Persistence Layer (Repository Layer)**

* Implements data access logic using Spring Data JPA.
* Contains repositories for each entity.
* Handles complex database queries and communicates with the business layer.

**Model / Data Layer**

* Defines entity classes that represent the core domain models (e.g., User, Venue, Booking, Payment, etc.).
* These are directly mapped to the relational database tables using JPA annotations.
* This layer ensures that your business logic operates on structured, validated, and consistent data.

This architecture aligns with modern software development best practices, enabling rapid development, easy testing, and future scalability.

## 3.2 Package diagram

A screenshot of a computer screen

AI-generated content may be incorrect.

## 3.3 Class diagram

A computer screen shot of a computer screen

AI-generated content may be incorrect.

## 3.4 Database (E-R/Data model) diagram

## *A screenshot of a computer program AI-generated content may be incorrect.*3.5 Sequence diagram

**Client Books a Venue:**

1.Client fills in booking details on the **frontend**.

2.The **BookingController** receives the request.

3.It calls **BookingService** to process it.

4.**BookingService**:

* Validates the booking.
* Checks venue availability.
* Saves the booking via **BookingRepository**.

5. Returns confirmation to frontend.

## A diagram of a software company AI-generated content may be incorrect.

## Client Makes a Payment:

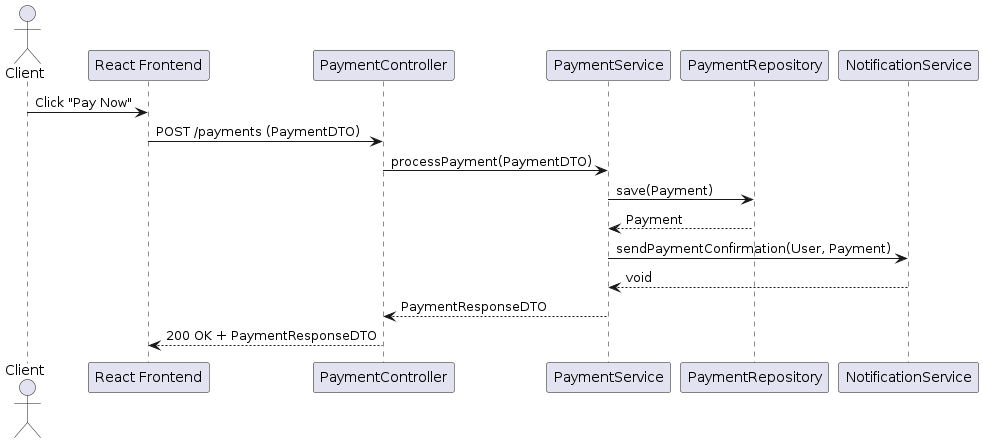
1.Client initiates payment after booking confirmation.

2.The payment request is handled by the **PaymentController**.

3.**PaymentService** processes the logic.

4.**PaymentRepository** saves the payment info.

5.Optionally, the service notifies the user of successful payment.



## 3.6 Activity diagram

Client Books an Event:

1. Client logs in
2. Views available venues
3. Selects venue and date
4. Enters event details (guest count, menu, etc.)
5. System checks availability
6. If unavailable → error message
7. If available → confirms booking and proceeds to payment
8. Ends with payment confirmation

A diagram of a process flow

AI-generated content may be incorrect.

# IV Supplementary specifications

The Supplementary Specification captures the system requirements that are not readily captured in the use cases of the use-case model. These include quality attributes, technical constraints, and other non-functional aspects of the system.

## 4.1 Non-functional requirements

**• Availability**

The EventEase platform is designed to be accessible 24/7 for both clients and administrators. Clients can access the system at any time to browse available venues, make bookings, or manage their events. Administrators can manage venues and events from anywhere, ensuring full-time support and flexibility in operations.

**• Performance**

The application will be optimized to handle simultaneous interactions by multiple users (admins and clients) without affecting performance. The backend is designed using efficient data access patterns and a layered architecture to minimize latency in user interactions, particularly during booking, calendar browsing, and event customization.

**• Security**

Security is a core aspect of EventEase. All user data is protected by a login system that requires a unique email and secure password. Passwords are encrypted using industry-standard hashing algorithms. Role-based access control ensures that only authorized administrators can view and modify sensitive event and client data. Additionally, critical changes by clients after a deadline require admin approval, adding another layer of operational control.

**• Usability**

EventEase offers a clean and intuitive interface for both clients and administrators. Clients are guided step-by-step through the event planning process, from selecting a date and venue to customizing every detail of their event. Administrators are provided with powerful tools for searching, updating, and managing bookings. The application is designed to reduce planning time and increase user satisfaction, even for first-time users.

**• Accessibility Across Devices**

As a web-based application, EventEase is accessible from any device with an internet connection and a modern browser. This makes it ideal for use on desktops, laptops, tablets, and even smartphones — whether by venue managers on-site or clients booking events remotely.

## 4.2 Design constraints

 The application will be developed using the **Java programming language**, following Object-Oriented Programming principles.

 The backend will use the **Spring Boot** framework for its layered architecture and **Hibernate (JPA)** for data persistence.

 The database will be implemented using **MySQL**, ensuring a relational and scalable data structure.

 The frontend will be built using **React**, chosen for its component-based structure and high interactivity, making it accessible for beginners and flexible for future extensions.

 The communication between frontend and backend will be handled through **RESTful API calls**, ensuring a clean separation of concerns and scalable integration.

 Email-based notifications will be implemented using **Spring Mail**, supporting booking confirmations, reminders, and admin notifications.(smtp server )

 Passwords will be hashed securely using **BCrypt**, and user authentication will be managed using **Spring Security**.

 The application will follow a **layered architectural style**, separating concerns into: controller, service, repository, model, and configuration layers to ensure maintainability and testability.

# V Testing

The testing strategy for the **EventEase** application focuses on validating business logic, ensuring security, and maintaining application stability as features evolve. Testing is primarily conducted on the **backend**, with plans to expand coverage to the **frontend** and user experience flows.

**Goals of Testing**

* Ensure the correctness of core features such as booking creation, user authentication, and event customization.
* Validate access control and role-based restrictions.
* Catch edge cases and prevent regressions during future updates.
* Provide maintainability and confidence in code quality through automated test suites.

**Current Coverage**

* **Unit Tests** for service-layer logic using **JUnit 5** and **Mockito**.
* **Integration Tests** using **Spring Boot Test** for repository and controller-level functionality.
* Manual exploratory testing of the frontend via browser-based interaction.

## 5.1 Testing methods/frameworks

The application uses **unit testing** and **integration testing** to ensure the reliability and correctness of backend logic. The following tools and frameworks are employed:

* **JUnit 5**: Used for writing and running unit tests across the service layer.
* **Mockito**: Used for mocking dependencies such as repositories and testing services in isolation.
* **Spring Boot Test**: Enables integration testing with full application context to simulate real API behavior.
* **MockMvc** (optional, if applicable): Used for testing REST endpoints without starting a full HTTP server.

Example test scenarios include:

* Verifying that bookings, venues, and users are correctly created, updated, or deleted.
* Asserting that role-based access restrictions are enforced.
* Testing edge cases such as null inputs, invalid IDs, or unauthorized access.

Test cases are located under the src/test/java/com/eventsapp/backend directory and follow a structured format for readability and maintainability.

## 5.2 Future improvements

While the current testing suite covers critical backend functionality, there are several areas for enhancement:

* **Frontend Testing**: Implement unit and integration tests for the React frontend using:
  + Jest and React Testing Library for component behavior and interaction testing.
* **End-to-End (E2E) Testing**: Add full-stack tests using Cypress or Selenium to simulate real user flows from login to event booking.
* **Test Coverage Reports**: Integrate tools like **JaCoCo** to measure test coverage and identify untested paths.
* **CI/CD Integration**: Automate test execution on each push or pull request using GitHub Actions, GitLab CI, or Jenkins.
* **Security Testing**: Incorporate basic tests to validate JWT expiration, access control, and input sanitization.

These improvements will increase application robustness, catch regressions early, and facilitate safer future feature development.

# VI Documentation of APIs

**GET /api/admin/bookings/awaiting-approval**

**Description:** Retrieve all bookings that are awaiting admin approval.

**Request:**  
● *No request body required*

**Responses:**  
● 200 OK – Returns a list of bookings with status AWAITING\_APPROVAL  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**GET /api/admin/bookings/manage**

**Description:** Retrieve all bookings that are neither confirmed nor rejected (pending admin action).

**Request:**  
● *No request body required*

**Responses:**  
● 200 OK – Returns a list of bookings with pending status  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**PUT /api/admin/bookings/{id}/approve**

**Description:** Approve a specific booking by ID and automatically create an event from it.

**Request:**  
● *Path Variable:* id – Long – ID of the booking to approve  
● *No request body required*

**Responses:**  
● 200 OK – Booking approved and event created  
● 404 NOT FOUND – Booking with the specified ID was not found  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**PUT /api/admin/bookings/{id}/mark-paid**

**Description:** Mark a specific booking as paid.

**Request:**  
● *Path Variable:* id – Long – ID of the booking to mark as paid  
● *No request body required*

**Responses:**  
● 200 OK – Booking marked as paid  
● 404 NOT FOUND – Booking with the specified ID was not found  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**PUT /api/admin/bookings/bookings/{id}/reject**

**Description:** Reject a specific booking by ID.

**Request:**  
● *Path Variable:* id – Long – ID of the booking to reject  
● *No request body required*

**Responses:**  
● 200 OK – Booking rejected  
● 404 NOT FOUND – Booking with the specified ID was not found  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**GET /api/admin/users**

**Description:** Retrieve a list of all registered users in the system.

**Request:**  
● *No request body required*

**Responses:**  
● 200 OK – Returns a list of users (as UserResponse objects)  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**DELETE /api/admin/users/{id}**

**Description:** Delete a specific user by their ID.

**Request:**  
● *Path Variable:* id – Long – ID of the user to delete  
● *No request body required*

**Responses:**  
● 204 NO CONTENT – User was successfully deleted  
● 404 NOT FOUND – User with the specified ID was not found  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**PUT /api/admin/events/{eventId}/update-price**

**Description:** Update the pricing for an event, including price per adult and per child.

**Request:**  
● *Path Variable:* eventId – Long – ID of the event to update  
● *Body (JSON):*

{

"pricePerAdult": 150.0,

"pricePerChild": 100.0

}

**Responses:**  
● 200 OK – Event pricing was successfully updated  
● 404 NOT FOUND – Event with the specified ID was not found  
● 400 BAD REQUEST – Invalid price values provided  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**GET /api/admin/events/{eventId}/full**

**Description:** Retrieve full details for a specific event, including menu, drinks, fruits, and miscellaneous settings.

**Request:**  
● *Path Variable:* eventId – Long – ID of the event to retrieve  
● *No request body required*

**Responses:**  
● 200 OK – Returns full event details as a FullEventDetailsResponse  
● 404 NOT FOUND – Event with the specified ID was not found  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**GET /api/admin/management/my-venues**

**Description:** Retrieve all venues and their scheduled events for the currently logged-in admin.

**Request:**  
● *Query Parameter:* email – String – Email of the admin whose venues should be retrieved  
● *No request body required*

**Responses:**  
● 200 OK – Returns a list of VenueWithEventsResponse containing venue and event details  
● 400 BAD REQUEST – Missing or invalid email parameter  
● 404 NOT FOUND – No venues or events found for the specified admin  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**PUT /api/admin/management/events/{eventId}/update-price**

**Description:** Update the calculated price per adult and child for a specific event.

**Request:**  
● *Path Variable:* eventId – Long – ID of the event to update  
● *Body (JSON):*

{

"pricePerAdult": 200.0,

"pricePerChild": 120.0

}

**Responses:**  
● 200 OK – Event pricing successfully updated  
● 400 BAD REQUEST – Invalid price data  
● 404 NOT FOUND – Event with the specified ID was not found  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**GET /api/admin/venues**

**Description:** Retrieve all venues in the system.

**Request:**  
● *No request body required*

**Responses:**  
● 200 OK – Returns a list of VenueResponse objects  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**POST /api/admin/venues**

**Description:** Create a new venue with optional image upload.

**Request:**  
● *Multipart Request:*

* data (JSON):

{

"name": "Venue Name",

"location": "City/Area",

"capacity": 200

}

* image (optional): JPG/PNG file

**Responses:**  
● 200 OK – Returns the created VenueResponse  
● 400 BAD REQUEST – Invalid data or file format  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**PUT /api/admin/venues/{id}**

**Description:** Update a venue by ID with optional new image.

**Request:**  
● *Path Variable:* id – Integer – ID of the venue to update  
● *Multipart Request:*

* data (JSON):

{

"name": "Updated Name",

"location": "Updated Location",

"capacity": 250

}

* image (optional): JPG/PNG file

**Responses:**  
● 200 OK – Returns the updated VenueResponse  
● 400 BAD REQUEST – Invalid input data or file  
● 404 NOT FOUND – Venue with specified ID not found  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**GET /api/admin/venues/{venueId}/available-admins**

**Description:** Retrieve a list of available admin users who can be assigned to the venue.

**Request:**  
● *Path Variable:* venueId – Integer – ID of the venue

**Responses:**  
● 200 OK – Returns a list of UserResponse objects  
● 404 NOT FOUND – Venue not found or no available admins  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**POST /api/admin/venues/{venueId}/assign-admins**

**Description:** Assign a list of admin users to a specific venue.

**Request:**  
● *Path Variable:* venueId – Integer  
● *Body (JSON):*

[1, 2, 3]

**Responses:**  
● 200 OK – Admins successfully assigned  
● 404 NOT FOUND – Venue or admin(s) not found  
● 400 BAD REQUEST – Invalid admin ID list  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**GET /api/admin/venues/my-venues**

**Description:** Retrieve the list of venues assigned to the currently authenticated admin.

**Request:**  
● *Header:* Authorization: Bearer <JWT\_TOKEN>  
● *No request body required*

**Responses:**  
● 200 OK – Returns a list of VenueResponse for the authenticated admin  
● 401 UNAUTHORIZED – Missing or invalid token  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**GET /api/admin/venues/venue/{venueId}/events**

**Description:** Retrieve all events scheduled at a specific venue.

**Request:**  
● *Path Variable:* venueId – Integer – ID of the venue

**Responses:**  
● 200 OK – Returns a list of Event objects  
● 404 NOT FOUND – Venue with specified ID not found  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**GET /api/admin/apetizers**

**Description:** Retrieve all available appetizers.

**Request:**  
● *No request body required*

**Responses:**  
● 200 OK – Returns a list of Apetizer objects  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**GET /api/admin/apetizers/{id}**

**Description:** Retrieve a specific appetizer by its ID.

**Request:**  
● *Path Variable:* id – Integer – ID of the appetizer

**Responses:**  
● 200 OK – Returns the Apetizer object  
● 404 NOT FOUND – Appetizer with specified ID not found  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**POST /api/admin/apetizers**

**Description:** Create a new appetizer (admin only).

**Request:**  
● *Body (JSON):*

{

"name": "Bruschetta",

"description": "Toasted bread with tomatoes",

"price": 15.5

}

**Responses:**  
● 200 OK – Appetizer successfully created  
● 400 BAD REQUEST – Invalid input data  
● 401 UNAUTHORIZED – If user is not logged in  
● 403 FORBIDDEN – If user lacks admin role  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**PUT /api/admin/apetizers/{id}**

**Description:** Update an existing appetizer (admin only).

**Request:**  
● *Path Variable:* id – Integer – ID of the appetizer to update  
● *Body (JSON):*

{

"name": "Updated Bruschetta",

"description": "Updated description",

"price": 17.0

}

**Responses:**  
● 200 OK – Appetizer successfully updated  
● 400 BAD REQUEST – Invalid input data  
● 404 NOT FOUND – Appetizer not found  
● 401 UNAUTHORIZED – If user is not logged in  
● 403 FORBIDDEN – If user lacks admin role  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**DELETE /api/admin/apetizers/{id}**

**Description:** Delete a specific appetizer by ID (admin only).

**Request:**  
● *Path Variable:* id – Integer – ID of the appetizer to delete

**Responses:**  
● 204 NO CONTENT – Appetizer successfully deleted  
● 404 NOT FOUND – Appetizer not found  
● 401 UNAUTHORIZED – If user is not logged in  
● 403 FORBIDDEN – If user lacks admin role  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**POST /api/auth/register**

**Description:** Register a new user (client, admin, or other role) and receive a JWT token upon success.

**Request:**  
● *Body (JSON):*

{

"name": "John Doe",

"email": "john.doe@example.com",

"password": "securePassword123",

"role": "CLIENT" // or "ADMIN"

}

**Responses:**  
● 200 OK – Returns an AuthResponse with a JWT token and user info

{

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6...",

"name": "John Doe",

"email": "john.doe@example.com",

"role": "CLIENT"

}

● 400 BAD REQUEST – Invalid or missing registration details  
● 409 CONFLICT – User with the same email already exists  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**POST /api/auth/login**

**Description:** Log in an existing user and return a JWT token upon successful authentication.

**Request:**  
● *Body (JSON):*

{

"email": "john.doe@example.com",

"password": "securePassword123"

}

**Responses:**  
● 200 OK – Returns an AuthResponse with a JWT token and user info

{

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6...",

"name": "John Doe",

"email": "john.doe@example.com",

"role": "CLIENT"

}

● 401 UNAUTHORIZED – Invalid email or password  
● 400 BAD REQUEST – Missing login details  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**POST /api/bookings**

**Description:** Create a new event booking by the currently authenticated client.

**Request:**  
● *Header:*  
Authorization: Bearer <JWT\_TOKEN>

● *Body (JSON):*

{

"venueId": 1,

"eventDate": "2025-07-15",

"startingTime": "18:00",

"estimatedGuestCount": 100,

"paidSum": 2000

}

**Responses:**  
● 200 OK – Booking created successfully  
● 400 BAD REQUEST – Invalid or incomplete booking data  
● 401 UNAUTHORIZED – If the user is not authenticated  
● 404 NOT FOUND – Venue not found or user not found  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**PUT /api/client/events/{eventId}/menu/custom**

**Description:** Assign a custom menu (appetizers, soups, main course) to an event by the client.

**Request:**  
● *Path Variable:* eventId – Long – ID of the event to customize  
● *Body (JSON):*

{

"appetizerIds": [1, 2],

"soupIds": [3],

"mainCourseId": 5

}

**Responses:**  
● 200 OK – Custom menu assigned to event  
● 400 BAD REQUEST – Invalid or incomplete menu data  
● 404 NOT FOUND – Event or menu item(s) not found  
● 403 FORBIDDEN – User is not authorized to modify this event  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

**PUT /api/client/events/{eventId}/misc/custom**

**Description:** Assign custom miscellaneous settings to an event (e.g., napkin color, decoration notes).

**Request:**  
● *Path Variable:* eventId – Long – ID of the event to customize  
● *Body (JSON):*

{

"napkinColor": "RED",

"otherDetails": "Please include floral table centerpieces."

}

**Responses:**  
● 200 OK – Custom miscellaneous settings assigned to event  
● 400 BAD REQUEST – Invalid miscellaneous data  
● 404 NOT FOUND – Event not found  
● 403 FORBIDDEN – User is not authorized to modify this event  
● 500 INTERNAL SERVER ERROR – If a server-side error occurs

# VIIBibliography

 **Spring Boot Documentation**  
<https://docs.spring.io/spring-boot/docs/current/reference/html/>  
– Used for building and structuring the backend REST API, dependency injection, and integrating Spring Security and JPA.

 **Spring Security Documentation**  
<https://docs.spring.io/spring-security/reference/>  
– Provided guidance on implementing role-based access control, JWT authentication, and securing endpoints.

 **JUnit 5 and Mockito Documentation**  
https://junit.org/junit5/  
https://site.mockito.org/  
– Referenced for creating unit and integration tests for service and controller layers.

 **React Documentation**  
https://reactjs.org/docs/getting-started.html  
– Used in building the frontend UI components and managing application state.

 **Material UI & MUI Documentation**  
https://mui.com/material-ui/  
– Utilized for designing a responsive and accessible user interface in the admin and client dashboards.

 **JWT.io – JSON Web Tokens**  
https://jwt.io/introduction  
– Provided foundational understanding for implementing secure token-based authentication.

 **Baeldung Tutorials**  
<https://www.baeldung.com/>  
– A helpful resource for Spring Boot, security, and REST API best practices.

 **OpenAPI Specification (Swagger)**  
https://swagger.io/specification/  
– Referenced for documenting REST endpoints and planning future API documentation improvements.

 **GitHub Repositories and Developer Forums**  
– Various discussions and examples helped troubleshoot implementation challenges and understand real-world applications of design patterns.