NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Wee Kim Wee School of Communication and Information
IN6206: Internet and Web Development
2025-2026 (Semester 1)
Individual Assignment Report

Eventor- An Event Management System

SUBMITTED BY

Balakrishnan Thoshinny (G2505848D)

1. Project Overview

Eventor is an event management platform designed to bridge the gap between event organizers (vendors) and attendees. The system provides a seamless digital experience for users to search events, book tickets, process payments, and receive automated notifications. Furthermore, vendors can post events, keep track of ticket bookings, and allow other CRUD operations. Built with modern full-stack technologies, Eventor ensures scalability, security, and user-friendliness.

2. Project Objectives & Scope

2.1 Primary Objectives

- Provide a user-friendly interface to search and book events.
- Enable vendors to create, manage, and analyze event performance.
- Integrate secure and reliable payment processing.
- Automate ticket generation and email notifications.
- Adopt a microservices-friendly architecture for modular development.
- Implement asynchronous communication using Kafka to avoid blocking operations.

2.2 Functional Scope

- User and vendor authentication and authorization.
- Event creation, editing, and deletion by vendors.
- Ticket booking, payment processing, and invoice generation.
- Automatic email delivery and PDF ticket generation through Kafka consumers.
- Containerized frontend, backend, Kafka, and database services.

2.3 Non-Functional Requirements

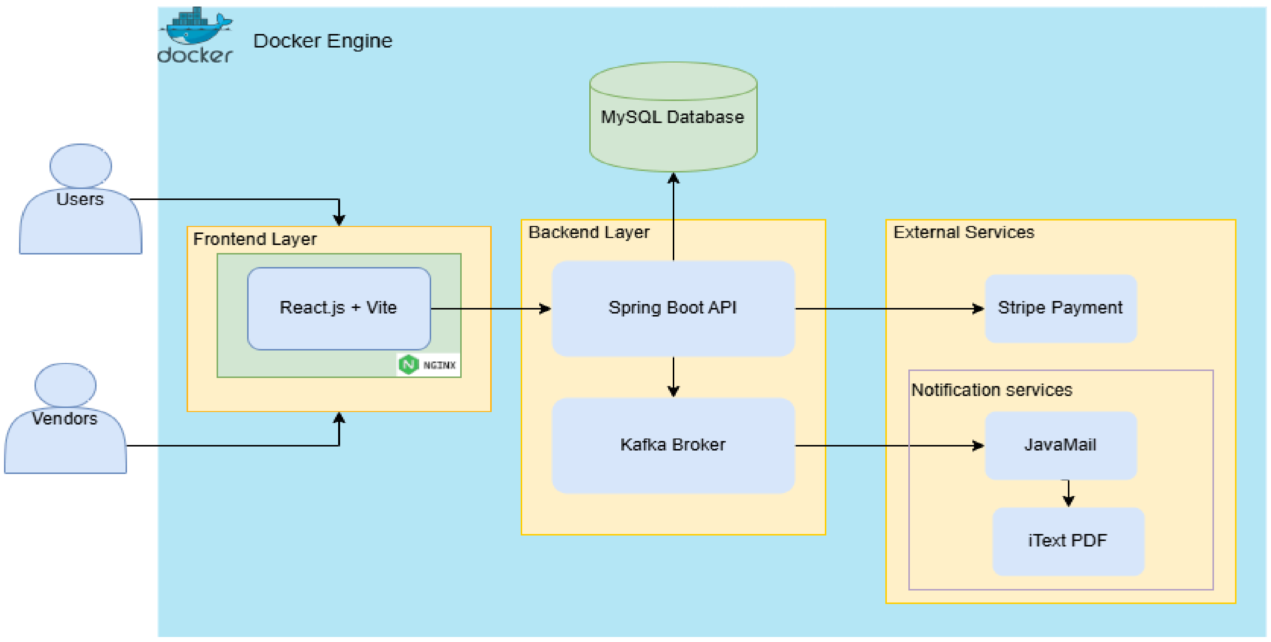
- Responsive and intuitive UI using React.js.
- Secure API endpoints using Spring Boot and Spring Security.
- Asynchronous and decoupled communication using Kafka.
- High maintainability and scalability using Docker-based microservices.
- SPA-compatible routing through custom Nginx configuration.
- Clean separation of concerns across frontend, backend, messaging, and database layers.

3. System Architecture Overview

Eventor follows a layered architecture

- Frontend Layer: React.js with Vite for fast rendering.
- Backend Layer: Spring Boot REST API handling business logic.
- External Services and Notification layer: Kafka for event-driven notifications and Stripe as a payment gateway.
- Data Layer: MySQL for persistent storage.
- Deployment: All components run as isolated containers connected via a Docker bridge network. Nginx exposes port 3000, backend exposes 8080, Kafka and Zookeeper run as independent services. This decoupling improves availability, scaling, and maintainability.

Users interact with the React frontend. Spring Boot processes requests, interacts with the database, and communicates with external services. Kafka decouples time-intensive tasks like email and PDF generation. Stripe handles all payment transactions securely.

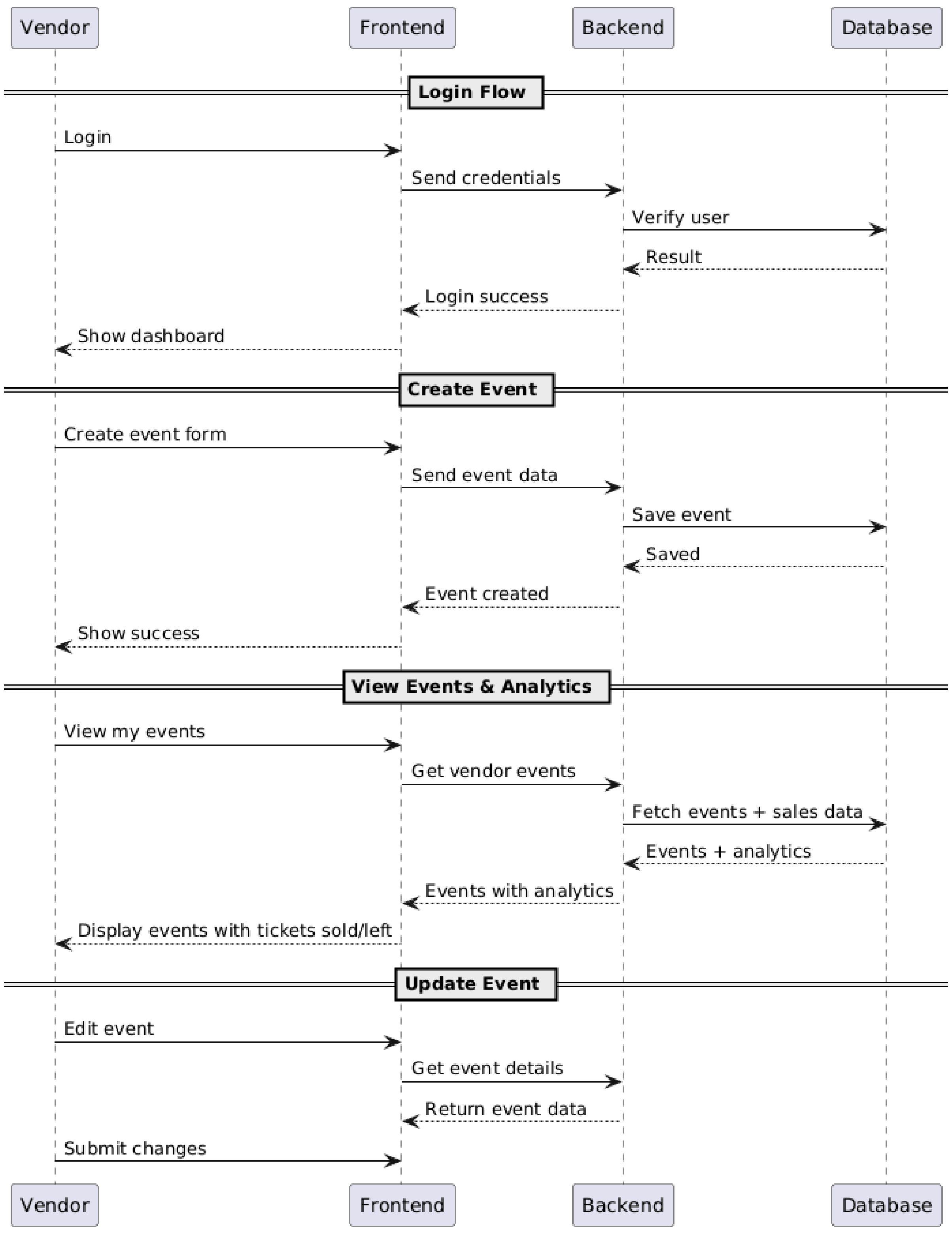


4. Technology Stack

Layer	Technologies Used
Frontend	React.js, Vite, Axios, JavaScript, Nginx
Backend	Spring Boot 3, Spring Security, Spring Data JPA, itextpdf, Javax mail
Messaging	Apache Kafka, Spring Kafka
Database	MySQL 8.0, Hibernate ORM
Payments	Stripe API
Notifications	JavaMail, iText PDF 7
Infrastructure	Docker, Docker Compose, Microservices Architecture
Development Tools	IntelliJ IDEA, VS Code, Postman, Git, Maven

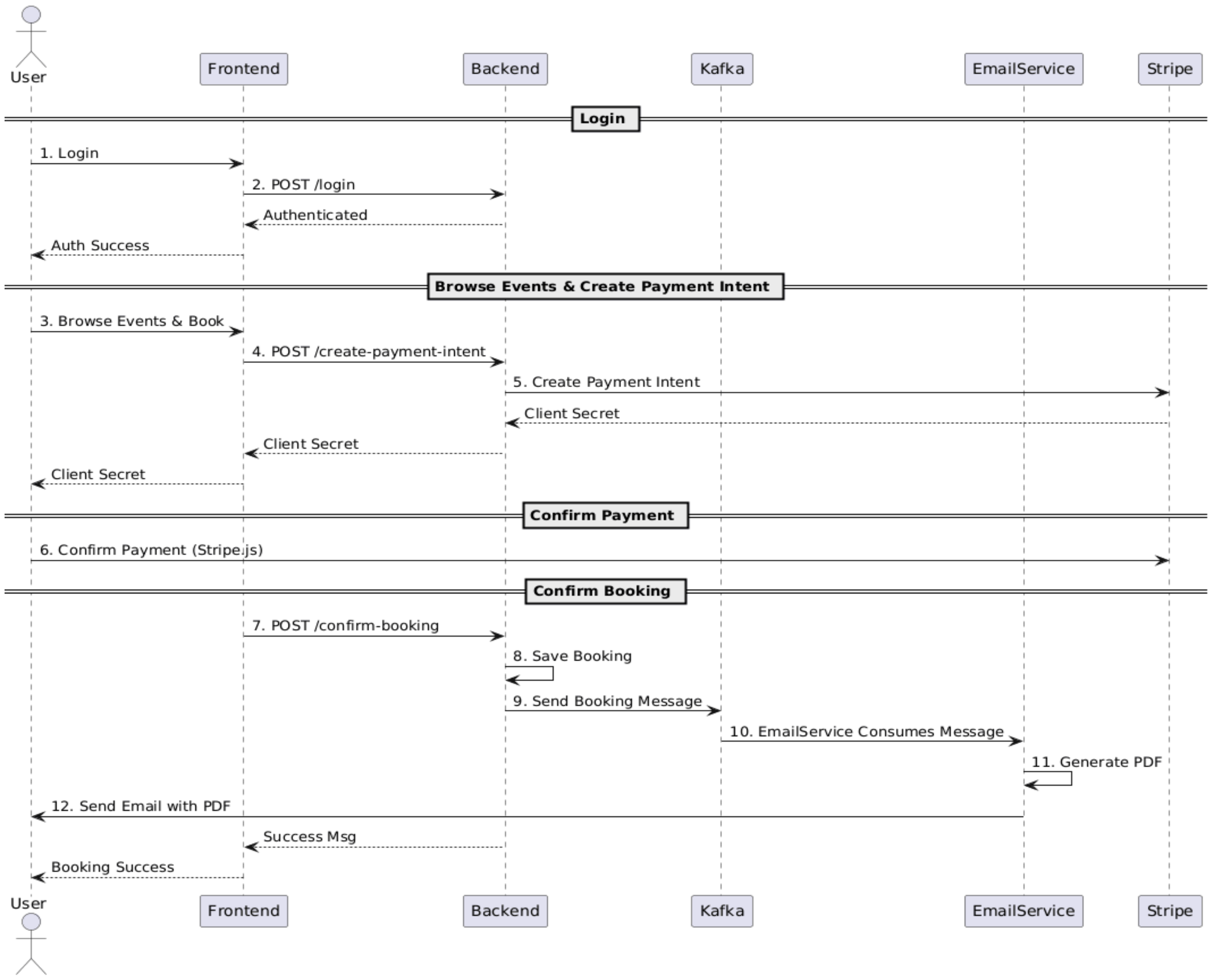
5. Vendor Workflow

- Login: Access the vendor dashboard.
- Create Event: Add event details, capacity, pricing, etc.
- Manage Events: Edit, delete, or view event analytics.
- Track Sales: Monitor tickets sold, revenue, and attendance.



6. User Journey

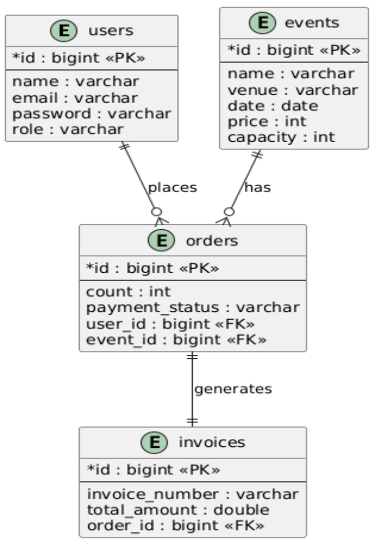
- Browse Events: Users view available events with search and filter options.
- Select Event: Choose an event, select ticket quantity, and proceed to checkout.
- Payment: Secure payment via Stripe.
- Confirmation: Receive email with PDF ticket attached.

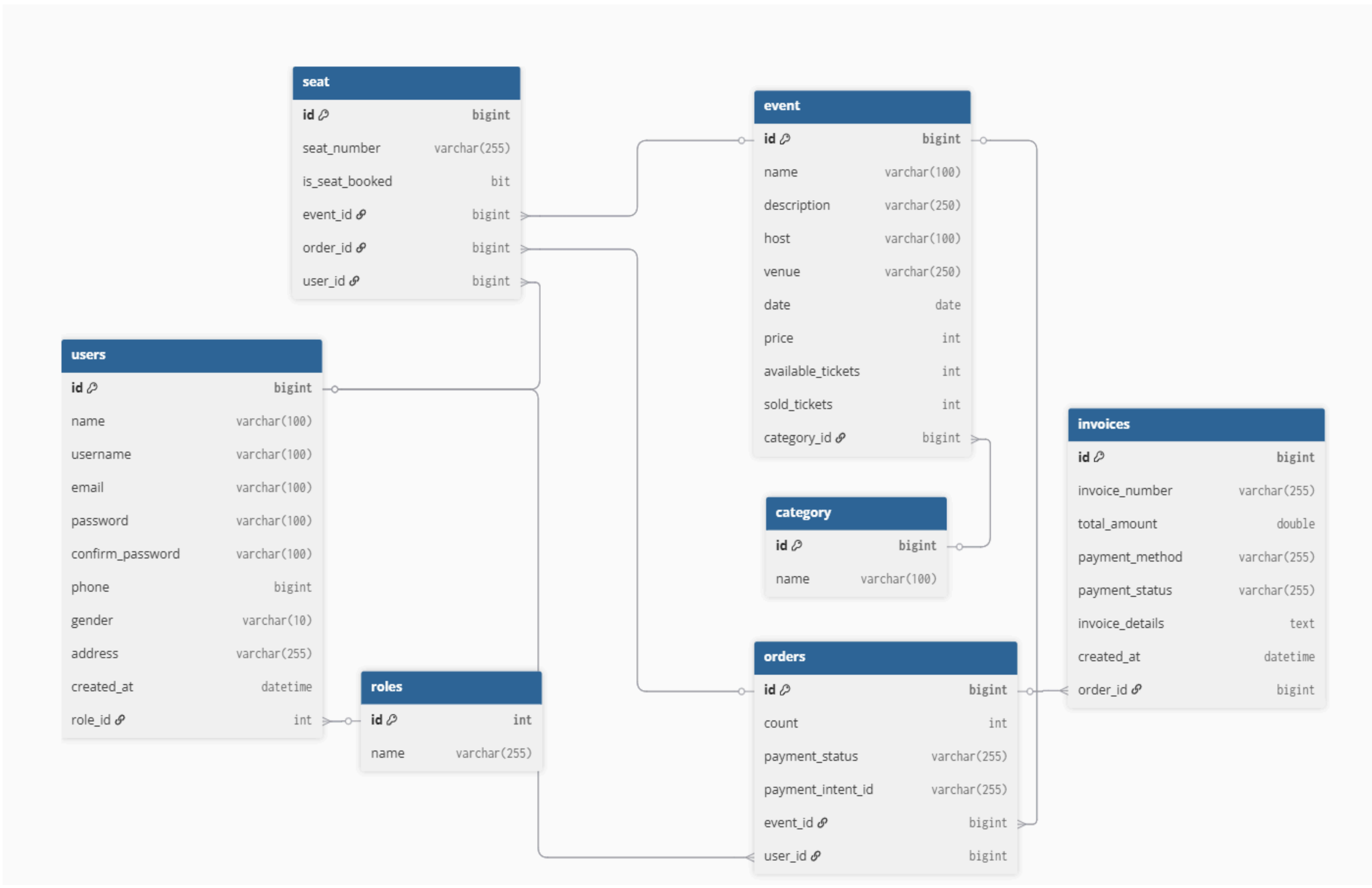


7. ER Diagram and Database Design

- **users:** Stores user and vendor accounts.
- **events:** Contains event details like name, date, venue, price, and capacity.
- **orders:** Records each booking with payment status.
- **invoices:** Stores payment and invoice details.
- **categories:** Manages event categories.

The database follows a relational model where users can place multiple orders for various events. Each event belongs to a category and maintains ticket capacity tracking. Orders generate invoices and can include multiple seat reservations for seated events. Foreign key constraints ensure data integrity across users, events, orders, and invoices relationships.





8. Design Patterns & Architecture Principles:

8.1 Design Patterns Used

- **DTO Pattern**: Used to transfer structured data between the frontend and backend. DTOs ensure that only necessary fields are exposed to the client by leveraging abstraction, improving security, reducing payload size, and maintaining a clean separation between internal entity models and external API contracts.

- **Publisher-Subscriber Pattern:** Implemented using Kafka, where the backend publishes booking events and a separate notification microservice subscribes to them. This decouples heavy operations, such as email sending and PDF generation, from the main request–response flow, improving scalability and responsiveness.

- **Layered Architecture:** The backend follows a classic Spring Boot layered architecture, consisting of Controller, Service, and Repository layers. This separation of concerns improves readability, maintainability, and testability. Each layer has a distinct responsibility, aligning the application with clean architecture principles.

- **Factory Pattern:** Used in the PDF generation module to construct dynamic ticket documents using iText. The factory pattern simplifies the creation of complex PDF objects such as ticket layouts, fonts, and templates while keeping the generation logic centralized.

- **Gateway/Reverse Proxy Pattern (via Nginx):** Nginx acts as an HTTP gateway that serves the frontend, handles routing, and optimizes asset delivery. It can also be extended in the future to manage load balancing and caching.

- **Client–Server Pattern**: The frontend and backend maintain strict boundaries, with well-defined REST APIs ensuring interoperability.

- **Singleton Pattern (Spring Beans):** Spring Boot manages service beans as singletons, ensuring shared resources (e.g., KafkaTemplate, EmailService) are efficiently reused across requests.

The decoupled microservice usage in Eventor aligns primarily with the Microservices Architecture Pattern, supported by the Event-Driven Architecture Pattern. Kafka enables a Publish–Subscribe model, ensuring that heavy tasks like email/PDF generation are handled asynchronously and independently. This combination provides strong scalability, loose coupling, resilience, and clear separation of responsibilities across services.

8.2 Spring Boot Features

- **Spring Security:** Provides robust authentication and authorization features, ensuring only verified users can access protected resources. It secures endpoints using filters, role-based access, JWT, and password encryption.

- **Spring Data JPA:** Simplifies database interactions by providing an abstraction over Hibernate ORM. It reduces boilerplate code using repositories, allowing easy CRUD operations and custom queries.

- **Spring Kafka:** Enables integration with Apache Kafka for asynchronous, event-driven communication between services. It supports message publishing, consuming, and error handling through simple KafkaTemplate and Listener APIs.

- **Spring Boot Starter mail:** Allows the application to send emails easily using JavaMail with minimal configuration. Used for sending notifications, attachments, and automated messages such as ticket PDFs.

9. Event-Driven Workflow

- Eventor adopts an event-driven architecture to handle asynchronous and time-consuming operations without blocking the user experience. When a user completes a booking, the backend Spring Boot service publishes a BookingCreatedEvent to a Kafka topic. This event contains essential details such as the user email, event information, payment confirmation, and booking metadata.

- A dedicated Notification Microservice—implemented using Spring Boot & Spring Kafka—listens to this topic as a consumer. Once the event is received, it initiates the generation of a PDF ticket using iTextPDF and sends the confirmation email via Spring Boot Starter Mail. Because these workflows run outside the main request-response cycle, the booking API returns instantly, improving responsiveness and preventing delays caused by file generation or external email services.

- This workflow demonstrates the core principles of event-driven systems: loose coupling, asynchronous processing, and independent horizontal scaling, making the system significantly more maintainable and resilient.

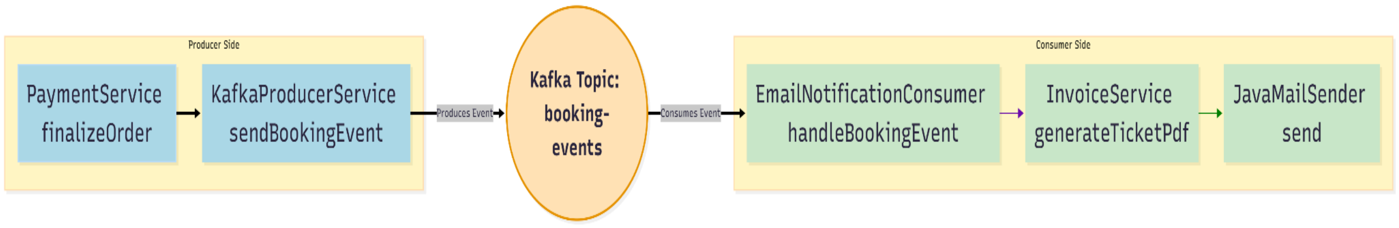
9.1 Benefits of this approach

- Scalability: Kafka allows multiple independent consumers to subscribe to the same topic, enabling services such as notifications, analytics, and audit logging to scale without modifying the booking module. Each consumer processes events at its own pace, ensuring high throughput even under heavy traffic.

- Decoupling: The booking service does not depend on email or PDF generation logic. Each microservice—Booking, Notification, Vendor, Payment—operates independently following the Publish–Subscribe design pattern, enabling modular development and reducing inter-service dependency.

- Resilience: If the notification service goes down or email sending temporarily fails, Kafka retains the event in its topic. Once the consumer recovers, it resumes processing without losing any booking notifications. This fault-tolerance ensures system reliability and protects critical workflows.

- Improved User Experience: End-users receive instantaneous confirmation after booking because the backend doesn't wait for PDF rendering or SMTP operations. Long-running tasks are executed asynchronously, keeping the application responsive.


Producer Side
PaymentService finalizeOrder
KafkaProducerService sendBookingEvent
Produces Event
Kafka Topic: booking-events
Consumes Event
Consumer Side
EmailNotificationConsumer handleBookingEvent
InvoiceService generateTicketPdf
JavaMailSender send

10. Challenges and Learnings

Payment Gateway Challenge
Razorpay doesn't support MVP projects, creating a payment integration blocker. Yet, pivoted to the Stripe API, which offered better global coverage. The migration required rewriting payment controllers and webhook handlers.

Dependency Version Conflicts
Spring Boot 3.x had deprecated methods, breaking existing code. We faced compatibility issues with Hibernate and security libraries. Solved by upgrading dependencies and refactoring affected repository classes.

Kafka Messaging Setup
Initial Kafka configuration caused message serialization errors. Consumer groups weren't properly reading booking events. Fixed by adjusting topic configurations and implementing proper error handling.

PDF Generation & Email
Dynamic ticket generation failed due to font and layout issues in iText. Email attachments were corrupted during MIME encoding. Resolved by using predefined templates and proper multipart email structuring.

API Response Format
Axios intercepted inconsistent response structures from backend APIs. Frontend components crashed while parsing nested error objects. Standardized response DTOs with status codes and message fields solved the issue.

ERR_CHUNKED_ENCODING (200) Ok response:
This occurred due to a network-level interruption while receiving the HTTP response. The backend successfully processed the request, but the response stream became partially corrupted or truncated during transmission. As a result, Axios failed to parse the response JSON and threw an error—even though the operation succeeded.

11. Future Enhancements

Performance & User Experience
Implement Redis caching for event data and user sessions to reduce database load. Add QR code generation to tickets for quick event entry and validation. Develop interactive analytics dashboards with charts for vendor insights.

Platform Expansion
Build a cross-platform mobile application using the React Native framework. Integrate social features, including event sharing and friend recommendations. Add multi-language support for international user accessibility.

Real-time Features
Implement WebSocket connections for live notifications and updates. Create real-time seat availability tracking during the booking process. Add live chat support for user assistance during events.

12. Key Learnings

Technical Integration
Learnt full-stack development, connecting React frontend with Spring Boot API Implemented event-driven architecture using Kafka for decoupled systems Gained expertise in secure payment processing with Stripe integration.

Development Practices
Established robust error handling across frontend and backend layers. Learned dependency management and version conflict resolution. Implemented proper authentication and authorization flows.

System Design
Understood microservices principles with asynchronous communication. Gained experience in PDF generation and email service integration. Mastered database design with proper relationships and query optimization.

13. Conclusion

Eventor successfully demonstrates a modern, scalable, and user-friendly event management system. The use of React.js, Spring Boot, Kafka, and Stripe provides a solid foundation for future enhancements. The project highlights the importance of clear architecture, effective debugging, and iterative development in building enterprise-grade applications.

14. References

Spilca, L. (2021). Understanding Spring Boot and Spring MVC. In Spring Start Here. Manning Publications Co. LLC.

Bejeck, B. (2024). Spring Kafka. In Kafka Streams in Action, Second Edition. Manning Publications Co. LLC.

RV, R. (2016). Spring microservices : build scalable microservices with Spring, Docker, and Mesos (1st edition). Packt Publishing.

Pandey, R., & Packt Publishing, publisher. (2022). Containerize Spring Boot CRUD App with Docker and Docker Compose. ([First edition].) [Video recording]. Packt Publishing.

Garcia, M. M., & Telang, T. (2023). Learn Microservices with Spring Boot 3: A Practical Approach Using Event-Driven Architecture, Cloud-Native Patterns, and Containerization (Third edition). Apress. https://doi.org/10.1007/978-1-4842-9757-5