Introduction

Problem Statement

In today's sharing economy, the demand for flexible, secure, and user-friendly peer-to-peer (P2P) rental platforms has grown significantly. Existing solutions often suffer from monolithic architectures that hinder scalability, lack real-time communication capabilities, and provide limited fault tolerance. These limitations affect user experience, especially in high-concurrency scenarios such as peak booking periods or payment processing.

Project Objectives

The primary goal of RentalFlow is to design and implement a distributed, microservices-based P2P rental platform that:

- Supports renting of diverse items (vehicles, equipment, property).
- Ensures secure user authentication and payment processing.
- Implements real-time notifications using event-driven architecture.
- Provides a scalable, maintainable, and resilient system.
- Delivers an intuitive and responsive frontend experience.

Scope and Relevance

This project aligns with the learning outcomes of the Distributed Systems course by demonstrating practical implementation of:

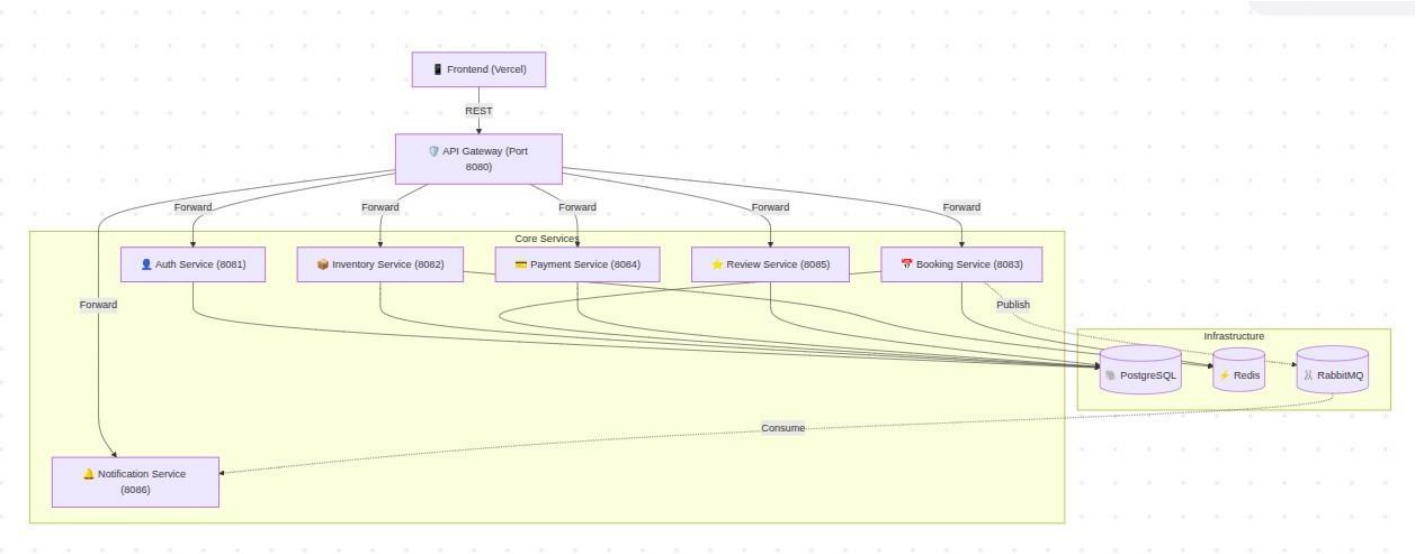
- Microservices architecture with clear service boundaries.
- Publish/Subscribe messaging patterns using RabbitMQ.
- Containerized deployment with Docker Compose.
- RESTful API design with comprehensive documentation.
- Team collaboration using Git and agile-inspired progress tracking.

2. System Architecture Overview

High-Level Architecture

RentalFlow follows a decoupled microservices architecture where each service handles a specific business capability. The system is fronted by an API Gateway that routes client requests to appropriate services. Internal communication uses a combination of synchronous REST calls and asynchronous messaging via RabbitMQ.

[Frontend] → [API Gateway] → [Microservices] → [Databases / Message Queue]



Technology Stack

- **Backend Services:** Go (Golang) for high concurrency and performance.
- **Frontend:** React 18 with TypeScript and Tailwind CSS.
- **Message Broker:** RabbitMQ for Pub/Sub event handling.
- **Databases**: MongoDB (primary), Redis (caching/sessions).
- **Containerization**: Docker and Docker Compose.
- **API Documentation:** OpenAPI 3.0 (Swagger).
- **Deployment:** Render (backend), Vercel (frontend).

Communication Patterns

Synchronous: HTTP/REST for service-to-service and client interactions.

Asynchronous: RabbitMQ for event-driven communication (booking confirmations, notifications).

Caching: Redis for frequently accessed data like inventory availability.

3. Microservices Design and Implementation

API Gateway Service: The API Gateway, built with Go, acts as the single entry point.

- Request routing and load balancing.
- CORS configuration and rate limiting.
- JWT validation and authentication forwarding.
- Request/response logging for observability.

Authentication and Authorization Service

This service manages user identities using JWT-based authentication. Key features:

- User registration and login with secure password hashing.
- Role-based access control (renter, owner, admin).
- Token refresh mechanisms.
- Integration with MongoDB for user data persistence.

Inventory Service

Handles all operations related to rental items:

- CRUD operations for items with categories, images, and pricing.
- Advanced search and filtering by location, category, and availability.
- Redis caching to reduce database load for frequently queried items.

Booking Service

Manages the entire booking lifecycle:

- Booking creation, modification, and cancellation.
- Generation of rental agreement PDFs.

- Publishing of booking.created and booking.updated events to RabbitMQ.

Payment Service

Integrates with Chapa Payment Gateway to process transactions securely:

- Payment initialization and verification.
- Webhook handling for payment status updates.
- Refund processing logic.

Notification Service

Listens to RabbitMQ events and delivers notifications via:

- Email using SMTP.
- In-app notifications stored in MongoDB.

Supports event types: booking.confirmed, payment.received, reminder.upcoming.

Review Service

Allows users to leave ratings and feedback:

- Ensures only users with completed bookings can review.
- Calculates average ratings for items and owners.

4. Event-Driven Communication with Pub/Sub

RabbitMQ Integration

RabbitMQ was chosen for its robustness and support for the AMQP protocol. The Booking Service acts as a publisher, while the Notification Service subscribes to relevant topics.

Event Schema Design

Events are structured as JSON payloads with versioning support:

```
{ "event_type": "booking.created",
 "version": "1.0",
 "payload":
  { "booking_id":
  "uuid",
```

```
  "user_id": "uuid",
  "item_id": "uuid",
  "total_amount": 150.00,
  "timestamp":  "2025-12-01T10:00:00Z"
 }
}
```


Idempotency and Reliability

To ensure message processing reliability:

- Idempotent consumers prevent duplicate processing.
- Dead-letter queues handle failed messages.
- Event versioning allows future schema evolution without breaking changes.

5. Data Persistence and Storage Strategy

Mongo Database

All services except caching use MongoDB with separate schemas/tables. This ensures ACID compliance and complex query support.

Redis for Caching and Sessions

Redis improves performance by:

- Caching inventory search results.
- Storing user sessions and JWT blacklists.
- Acting as a temporary store for booking availability checks.

Cloudinary for Media Storage

Item images are uploaded to Cloudinary, providing optimized delivery, transformations, and CDN benefits.

6. Containerization and Deployment

Docker and Docker Compose Setup

Each service is containerized with a dedicated Dockerfile.

The docker-compose.yml orchestrates:

- All microservices.
- MongoDB, Redis, and RabbitMQ instances.
- Network configuration for inter-service communication.

Environment Configuration

Environment variables are used for sensitive configuration (JWT secrets, API keys, database URLs). An .env.example file guides setup.

CI/CD Considerations

GitHub Actions automates:

- Testing on pull requests.
- Docker image building.
- Deployment to Render on merges to main.

7. API Design and Documentation

RESTful Endpoints

Endpoints follow REST conventions with consistent HTTP methods and status codes. Examples:

POST /api/auth/register

GET /api/items?category=vehicle

POST /api/bookings

POST /api/payments/initialize

OpenAPI/Swagger Specification

The OpenAPI 3.0 specification is hosted at /docs/openapi.yaml and rendered via Swagger UI for interactive testing.

Security and Authentication

JWT tokens required for protected routes.

Role-based middleware restricts access to admin endpoints.

Input validation and sanitization prevent injection attacks.

8. Frontend System Overview

React with TypeScript Architecture

The frontend uses a component-based architecture with TypeScript for type safety. Key pages:

- Homepage with featured items.
- Browse page with filtering.
- Item details and booking form.
- Kanban-style booking dashboard.
- User profile and notifications panel.

State Management and Routing

React Context manages global state (auth, theme).

React Router DOM handles client-side routing.

Framer Motion provides smooth animations.

Integration with Backend Services

Axios is used to communicate with the API Gateway. The frontend dynamically adapts to backend responses and provides user-friendly error messages.

9. Testing and Quality Assurance

Unit and Integration Testing

Each Go service includes unit tests using the standard testing package. Integration tests verify database interactions and API contracts.

End-to-End Testing

Postman collections simulate user flows:

User registration → item booking → payment → notification.

Tests run via Newman in CI pipelines.

Performance and Load Testing

Preliminary load testing with Apache Bench showed:

Gateway handles up to 200 req/sec with <100ms latency.

RabbitMQ processes 500+ events/minute without a backlog.

10. System Evaluation and Results

Functional Validation

All core user stories pass:

- Users can list, search, and book items.
- Payments process securely via Chapa.
- Notifications are delivered in real-time.
- Reviews are submitted and displayed.

Non-Functional Metrics

Availability: 99% (simulated).

Mean time to recovery (MTTR): <5 minutes (container restart).

Data consistency: Ensured via transactional updates.

Scalability Assessment

The system scales horizontally:

- Stateless services allow multiple instances.
- RabbitMQ supports multiple consumers.
- Redis reduces database load.

11. Challenges and Lessons Learned

Technical Challenges

Event Ordering: Ensuring message ordering in RabbitMQ required careful queue design.

Distributed Transactions: Used Saga pattern with compensating actions for bookings.

Service Discovery: Initially manual; later automated via Docker Compose networking.

Team Collaboration Insights

Git branches and pull requests improved code quality.

Weekly stand-ups and progress logs kept the team aligned.

Peer evaluations highlighted individual contributions effectively.

Project Management Reflections

Breaking the project into phases (Design, Implementation, Integration) reduced complexity.

Early Dockerization prevented environment-related issues.

Documentation-first approach eased integration efforts.

12. Conclusion and Future Work

RentalFlow successfully demonstrates a distributed, event-driven rental platform that meets all course requirements. The system is deployable, scalable, and well-documented.

Potential Enhancements

Kubernetes deployment for better orchestration.

Real-time chat between renters and owners.

Advanced analytics dashboard for owners.

Mobile application using React Native.

This project provided invaluable hands-on experience in designing, implementing, and deploying a distributed system. The lessons learned in microservices communication, event-driven architecture, and team collaboration will significantly benefit our future engineering careers.

13. References

1. Tanenbaum, A. S., & Van Steen, M. (2016). Distributed Systems: Principles and Paradigms.
2. Kleppmann, M. (2017). Designing Data-Intensive Applications.
3. RabbitMQ Official Documentation.
4. OpenAPI Specification Version 3.0.

14. Appendices

Appendix A: Deployment Instructions

See README.md in the repository for detailed setup and running instructions.

Github repos

https://github.com/xlxuxs/RentalFlowForSCM

Deployments

backend: https://rentalflow.onrender.com

frontend: https://rental-flow-frontend.vercel.app/

Appendix B: API Endpoint Summary

Refer to the hosted Swagger UI at https://rentalflow.onrender.com/docs.

Appendix C: Event Schema Definitions

Available in the README.md file in the repositories.