School of Computational Sciences and Artificial Intelligence

Spring 2025

TicketOrLeaveIt

Team Members:

Abdelrahman elkady

Shahd Hossam

Kareem Adel

Rana Hazem

1. Introduction

1.1 Vision

We are building TicketOrLeaveIt, an event ticketing platform that connects organizers with attendees. The system will use Go for the backend and PostgreSQL for data storage to create a fast, reliable service.

1.2 Problem Statement

Current ticketing systems have problems with handling large crowds, lack good tools for organizers, and often break during busy times. Our solution will handle heavy traffic, give organizers better controls, and stay stable during peak usage.

1.3 Scope

The system will include:

- User accounts and secure login
- Event creation and management tools
- Ticket purchasing system
- Organizer dashboard
- System monitoring and error handling

---

2. Functional Requirements

For Users:

1. Users should be able to create an account using their first name, last name, email, and password.
2. Users should be able to update their profile information (name, profile picture, password).
3. Users should be able to view their event history.

4. Users should be able to log into their created accounts. Token-based authentication should be implemented using OAuth.

For Organizers:

5. The system should allow event organizers to create a new event using the following: event name, event type, description, date and time, location, capacity, ticket pricing, and event promotional images.
6. The system should allow event organizers to view the details and check the remaining capacity of an event using its event ID.
7. The system should allow organizers to categorize events by type (concert, conference, workshop) and add tags for easier searchability.
8. The system should send notifications to users for ticket purchase confirmations and event reminders.
9. Organizers should be able to view a dashboard displaying event performance metrics.

System Features:

10. The system should support centralized logging using OpenSearch.

---

3. Technical Design

System Structure

The application will be built as separate Microservices:

- Authentication service for user login
- User service for profiles
- Event service for event management
- Ticket service for purchases
- Notification service for emails

Technologies

- Programming Language: Go
- Database: PostgreSQL
- Additional Tools: Redis, Nginx, RabbitMQ

---

4. Non-Functional Requirements

The system should:

1. Be able to serve 2,000 users daily, up to 400 concurrent RPS at peak load.
2. Ensure data consistency and reliability, preventing data loss and ensuring all transactions are correctly processed.
3. Offer high throughput with low latency for its users.
4. Maintain an uptime of 99% or higher.

---

5. ADRs

ADR 001: Microservices Architecture

Status: Proposed

Context:
To manage a complex system like an event management and ticket booking platform, we need a scalable and fault-tolerant architecture.

Decision:
We will adopt a microservices architecture, with dedicated services for authentication, notification, users, and events.

Rationale:

- **Scalability:** Allows each service to be scaled independently based on demand.

- **Fault Tolerance:** Enhances fault tolerance by isolating failures to specific services.
- **Development Simplification:** Simplifies development and deployment by breaking down the system into smaller, manageable components.

Consequences:
Positive Impacts:

- Each service can be scaled independently based on demand.
- Failures are isolated to specific services, enhancing fault tolerance.
- Development and deployment are simplified by breaking down the system into smaller components.

Potential Drawbacks:

- Managing a microservices architecture may introduce additional complexity in terms of inter-service communication, monitoring, and deployment.
- Ensuring consistency and coordination across services may require additional effort and tools.

ADR 002: Choosing GoLang as the Main Coding Language

Status: Accepted

Context:
Our system is built on a microservices architecture, requiring a programming language that is efficient, scalable, and well-suited for distributed systems. We need to choose a language that aligns with our goals of high performance, ease of development, and maintainability.

Decision:
We will use Go (Golang) as the primary programming language for developing our microservices.

Rationale:

- **Performance:** Go is a statically typed, compiled language known for its high performance and low latency, making it ideal for handling high-concurrency scenarios common in microservices.

- **Simplicity:** Go's syntax is simple and easy to learn, reducing the learning curve for developers and speeding up development.
- **Concurrency Support:** Go has built-in support for concurrency through goroutines and channels, which are essential for handling multiple requests simultaneously in a microservices architecture.
- **Scalability:** Go's lightweight goroutines and efficient memory management make it highly scalable, allowing us to handle large volumes of traffic with minimal resource usage.
- **Strong Standard Library:** Go comes with a robust standard library that includes packages for HTTP servers, JSON parsing, and more, reducing the need for third-party dependencies.
- **Community and Ecosystem:** Go has a growing community and a rich ecosystem of libraries and tools, making it easier to find solutions and best practices for microservices development.

Consequences:
Positive Impacts:

- High performance and low latency, ensuring that our microservices can handle high traffic and concurrent requests efficiently.
- Simplified development process due to Go's clean syntax and strong standard library.
- Built-in concurrency support enables efficient handling of multiple requests, improving system responsiveness.
- Scalability and efficient resource usage make Go a cost-effective choice for deploying microservices.
- Cross-platform support simplifies deployment across different environments.

Potential Drawbacks:

- Go's simplicity may limit flexibility in certain advanced use cases compared to more feature-rich languages like Python or Java.

- The Go ecosystem, while growing, may not have as many third-party libraries or frameworks as more established languages.

- Go is a relatively new programming language, which makes developers may require more time to learn the language and its concurrency model.

ADR 003: Rate Limiting (Sliding Window)

Status: Proposed

Context:
To prevent abuse of our system such as brute force attacks and DDoS attacks, we need to implement rate limiting.

Decision:
We will use the Sliding Window algorithm for rate limiting, leveraging Redis for its fast in-memory data storage capabilities.

Rationale:

- **Accuracy:** The Sliding Window algorithm is more accurate than fixed window or token bucket algorithms because it tracks requests in real-time over a rolling time window.

- **Scalability:** Redis supports distributed systems, making it scalable for high-traffic scenarios like ticket booking systems.

Consequences:
Positive Impacts:

- Protects our APIs and services from being overwhelmed by too many requests.

- Ensures fair usage of resources, especially during high-demand events like ticket sales.

- Prevents malicious users from exploiting the system.

Potential Drawbacks:

- Implementing and maintaining the Sliding Window algorithm may require additional development effort and computational resources.

- Redis, while fast, may introduce additional complexity in managing in-memory data storage and ensuring its availability.

ADR 004: Fault Tolerance (Graceful Degradation)

Status: Proposed

Context:
To ensure our system remains operational even when some components fail or are under heavy load, we need to implement graceful degradation.

Decision:
We will implement graceful degradation to maintain partial functionality during failures. This ensures that users can still interact with the system even if certain services (e.g., event search, ticket management) experience issues.

Rationale:

- **Operational Continuity:** If critical services fail, the system will continue to operate in a degraded mode, allowing users to browse events and perform essential actions.

- **User Experience:** The system will display a default list of events or provide fallback options to ensure users can still access key functionalities.

- **Fault Tolerance:** Graceful degradation ensures that the system remains usable, even if some components are unavailable or under heavy load.

Consequences:
Positive Impacts:

- Maintains partial functionality during failures, ensuring users can still browse events, view event details, and purchase tickets.

- Reduces the impact of failures on the overall user experience by providing fallback mechanisms.

- Provides time for recovery without completely shutting down the system.

Potential Drawbacks:

- Implementing graceful degradation may require additional logic and fallback mechanisms, increasing system complexity.
- Users may experience a reduced level of service or functionality during degraded mode, which could impact satisfaction.

ADR 005: Load Balancing (Nginx)

Status: Proposed

Context: To distribute incoming traffic across multiple servers and ensure no single server is overwhelmed, we need to implement load balancing.

Decision: We will use Nginx for load balancing and Redis for session storage and caching.

Rationale:

- **Performance:** Nginx is a high-performance load balancer that can handle thousands of concurrent connections.
- **Scalability:** Redis supports distributed caching, improving performance and scalability.

Consequences:

Positive Impacts:

- Distributes traffic evenly across multiple servers, preventing any single server from becoming a bottleneck.
- Improves system performance and reduces response times.
- Ensures high availability by rerouting traffic away from unhealthy servers.

Potential Drawbacks:

- Setting up and configuring Nginx and Redis for load balancing and session management may require additional expertise and effort.

- Redis, while efficient, may introduce additional complexity in managing distributed caching and ensuring data consistency.

ADR 006: Messaging & REST API

Status: Proposed

Context: To enable communication between microservices in a distributed system, we need to implement messaging and REST APIs.

Decision: We will use RabbitMQ for asynchronous communication and REST APIs for standardized service communication.

Rationale:

- **Decoupling:** Enables decoupling of services, making the system more modular and easier to maintain.
- **Fault Tolerance:** Improves fault tolerance by allowing services to continue operating even if one service fails.
- **Asynchronous Processing:** Supports asynchronous processing for tasks that do not need to be handled in real-time.

Consequences:

Positive Impacts:

- Makes the system more modular and easier to maintain.
- Enhances fault tolerance by isolating failures to specific services.
- Supports asynchronous processing for tasks like sending emails.

Potential Drawbacks:

- Implementing and managing RabbitMQ for asynchronous communication may introduce additional complexity.

- REST APIs, while standardized, may require careful design to ensure consistency and performance across services.

ADR 007: Database Selection (PostgreSQL)

Status: Proposed

Context: Our microservices architecture requires a database solution that ensures data consistency, availability, and proper isolation between services.

Decision: We will implement dedicated PostgreSQL databases for each microservice, with no replication between instances.

Rationale:

- **Reliability:** PostgreSQL is known for its reliability and support for complex queries.

- **Fault Isolation:**Prevents Single Point of Failure (SPOF) by decoupling service databases.

- **Performance:** Handles complex queries efficiently for our event and ticket management needs.

- **Operational Simplicity:** Avoids the complexity of managing a replicated database cluster.

- **Cost-Effectiveness:** Single instance deployment reduces infrastructure costs.

- **Data Consistency:** ACID compliance within each service boundary.

Consequences:

Positive Impacts:

- Eliminates database-level SPOF across services
- Ensures strong data consistency during peak loads
- Provides clear fault containment boundaries
- Supports complex transactions within each domain

- Enables service-specific optimization
- Simplifies debugging with isolated data stores

Potential Drawbacks:

- Requires careful retry logic implementation
- Service-specific maintenance downtime
- Temporary write unavailability during outages
- May need Redis caching for read resilience
- Requires performance tuning for scale

ADR 008: Circuit Breaker Pattern Implementation

Status: Proposed

Context: Our microservices architecture needs to handle failures gracefully and prevent cascading failures when dependent services become unavailable or exhibit high latency.

Decision: We will implement the Circuit Breaker pattern for inter-service communication to improve system resilience.

Rationale:

- Prevents cascading failures across microservices.

- Reduces load on failing services.

- Provides predictable failure modes.

- Faster failure detection than timeouts alone.

Consequences:

Positive Impacts:

- Improved system stability during partial outages.

- Better user experience with clear error messages.

- Reduced resource waste on doomed requests.

- Automatic recovery detection.

- Gives failing services time to recover.

Potential Drawbacks:

- Additional complexity.
- Requires careful tuning of thresholds.
- May mask underlying problems if misconfigured.
- Need for comprehensive monitoring.

ADR 009: Retry Logic Implementation

Status: Proposed

Context: Transient failures in distributed systems require resilient retry mechanisms to maintain service reliability without overwhelming failing components.

Decision: We will implement exponential backoff retry logic with jitter for all external service calls and database operations.

Rationale:

- Handles transient failures automatically.
- Exponential backoff prevents thundering herds.
- Clear retry eligibility prevents harmful retries.
- Balanced approach between persistence and responsiveness.

Consequences:

Positive Impacts:

- Increased success rates for transient failures.
- Better resource utilization during outages.

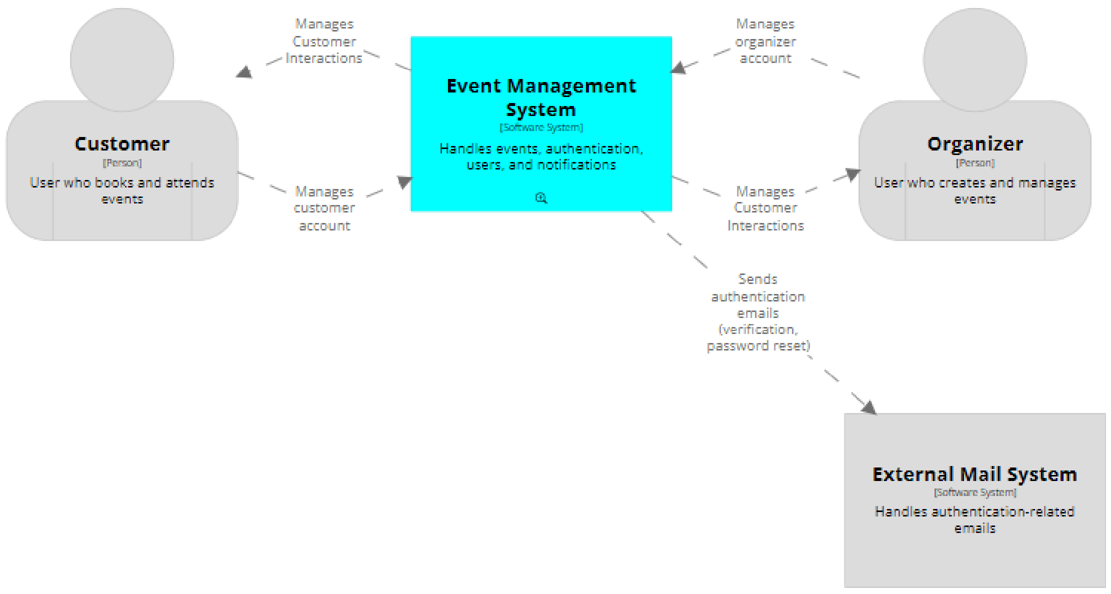
- Automatic recovery without user intervention.
- Configurable for different service needs.

Potential Drawbacks:

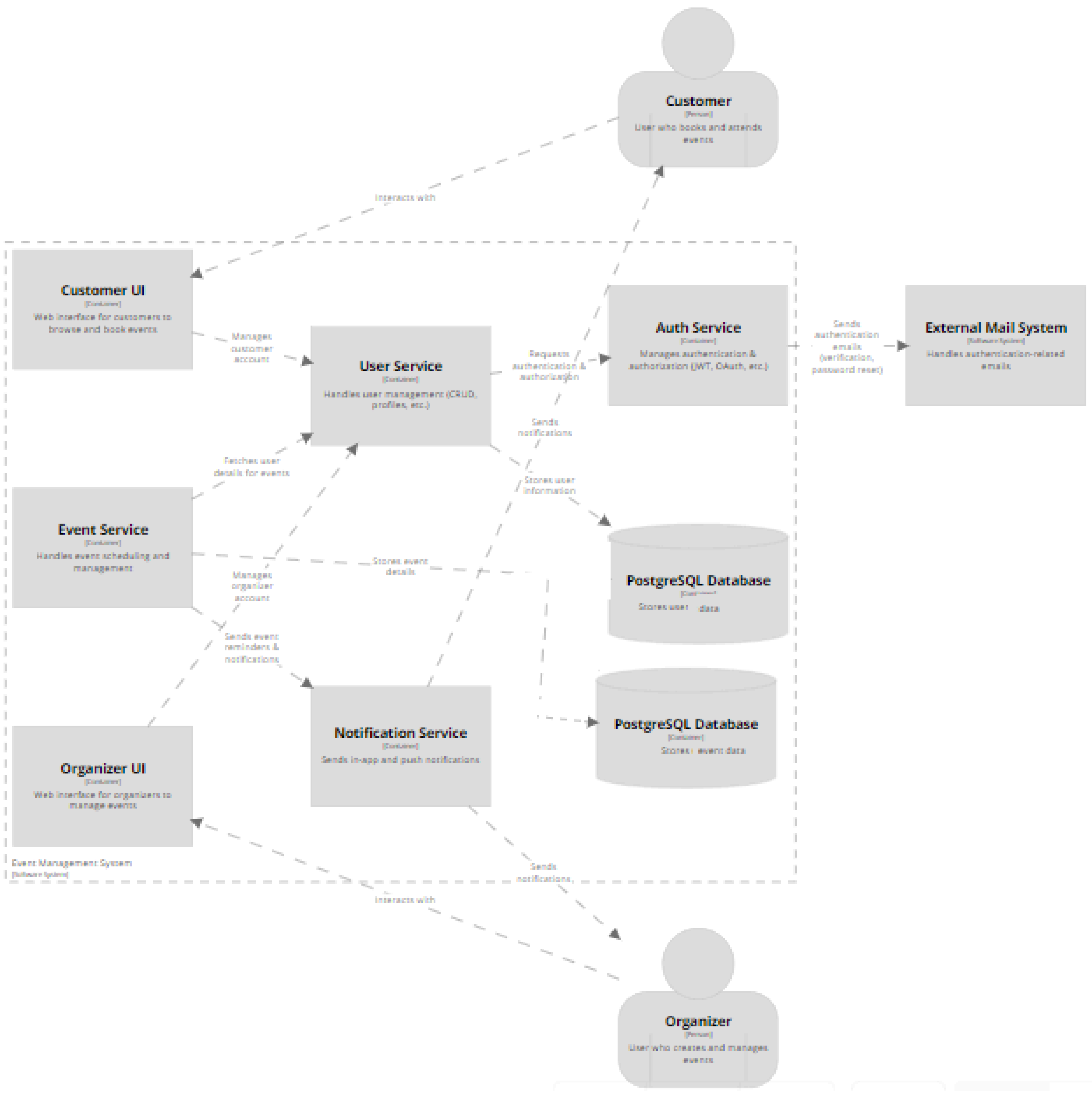
- Increased latency for end-users during retries.
- Potential for duplicate operations if not properly implemented.
- More complex error handling paths.

6. Diagrams

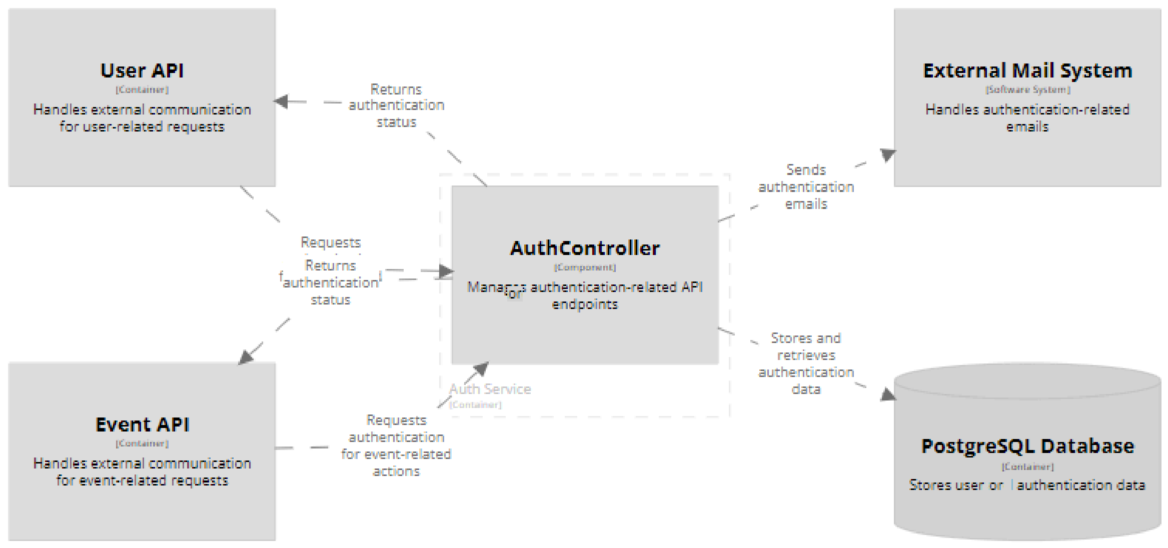
C1


Customer
[Person]
User who books and attends events
Manages Customer Interactions
Manages customer account
Event Management System
[Software System]
Handles events, authentication, users, and notifications
Manages organizer account
Manages Customer Interactions
Organizer
[Person]
User who creates and manages events
Sends authentication emails (verification, password reset)
External Mail System
[Software System]
Handles authentication-related emails

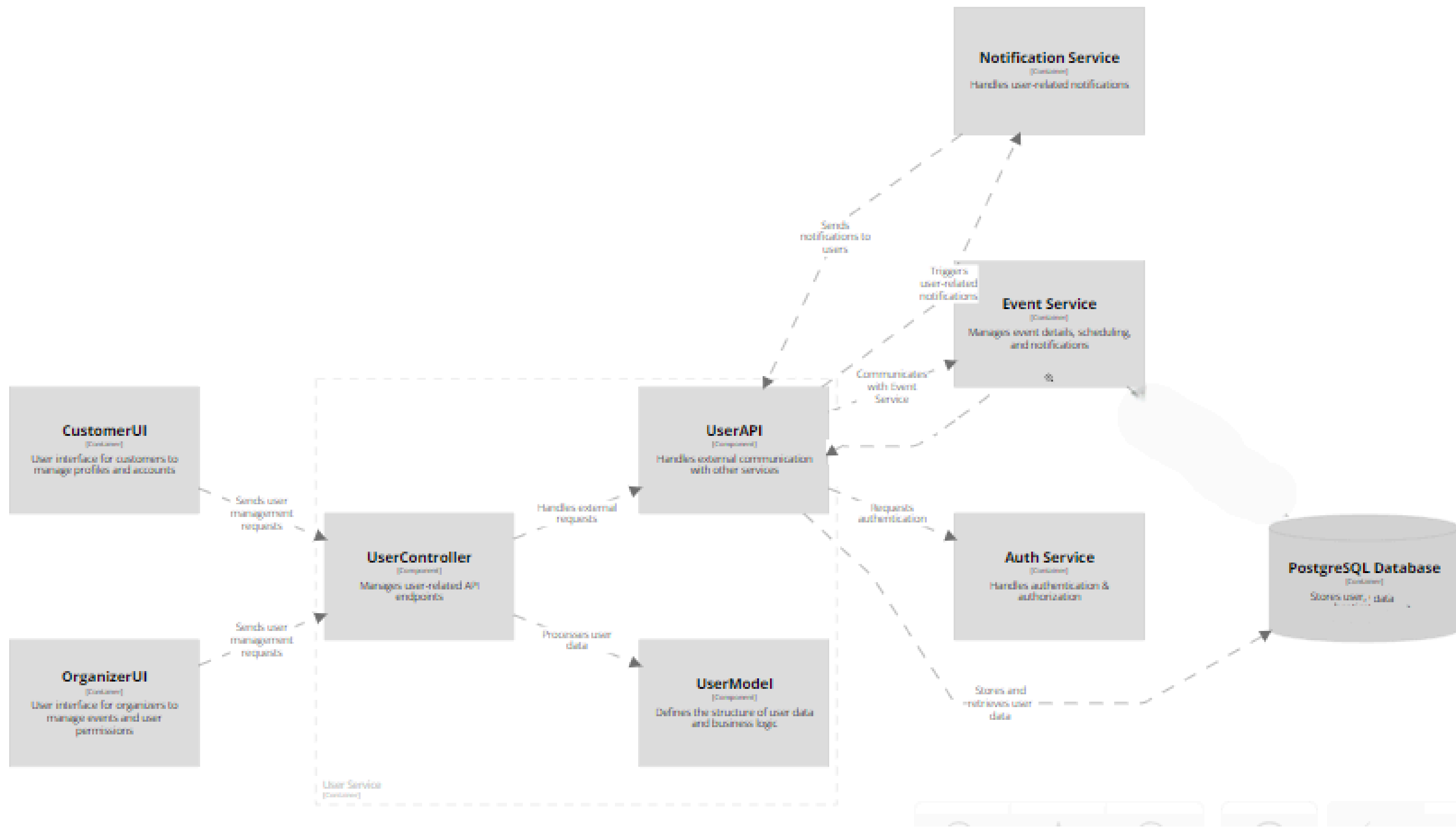
C2


Customer
[Person]
User who books and attends events
Interacts with
Customer UI
[Container]
Web interface for customers to browse and book events
Manages customer account
User Service
[Container]
Handles user management (CRUD, profiles, etc.)
Requests authentication & authorization
Auth Service
[Container]
Manages authentication & authorization (JWT, OAuth, etc.)
Sends authentication emails (verification, password reset)
External Mail System
[Software System]
Handles authentication-related emails
Sends notifications
Fetches user details for events
Stores user information
Event Service
[Container]
Handles event scheduling and management
Stores event details
PostgreSQL Database
[Container]
Stores user data
Manages organizer account
Sends event reminders & notifications
PostgreSQL Database
[Container]
Stores event data
Notification Service
[Container]
Sends in-app and push notifications
Organizer UI
[Container]
Web interface for organizers to manage events
Event Management System
[Software System]
Sends notifications
Interacts with
Organizer
[Person]
User who creates and manages events

C3 Auth


User API
[Container]
Handles external communication for user-related requests
Returns authentication status
External Mail System
[Software System]
Handles authentication-related emails
Sends authentication emails
Requests
Returns authentication status
AuthController
[Component]
Manages authentication-related API endpoints
Stores and retrieves authentication data
Auth Service
[Container]
Event API
[Container]
Handles external communication for event-related requests
Requests authentication for event-related actions
PostgreSQL Database
[Container]
Stores user or authentication data

C3 Users


Notification Service
[Container]
Handles user-related notifications
Sends notifications to users
Triggers user-related notifications
Event Service
[Container]
Manages event details, scheduling, and notifications
Communicates with Event Service
CustomerUI
[Container]
User interface for customers to manage profiles and accounts
Sends user management requests
UserAPI
[Component]
Handles external communication with other services
Handles external requests
UserController
[Component]
Manages user-related API endpoints
Requests authentication
Auth Service
[Container]
Handles authentication & authorization
PostgreSQL Database
[Container]
Stores user data
OrganizerUI
[Container]
User interface for organizers to manage events and user permissions
Sends user management requests
Processes user data
UserModel
[Component]
Defines the structure of user data and business logic
Stores and retrieves user data
User Service
[Container]