EvenToNight
Distributed Systems - Final Report

Alfonsi Alice (alice.alfonsi2@studio.unibo.it)
Bravetti Federico (federico.bravetti2@studio.unibo.it)
Brini Tommaso (tommaso.brini@studio.unibo.it)

March 2026

Abstract

The project consists of the design and development of a distributed digital platform called **EvenToNight**, aimed at connecting organizations that promote social events with users interested in discovering and participating in them. The platform, accessible at EvenToNight, can be used either as a guest or through a registered account that provides additional functionalities.

From a technical perspective, EvenToNight has been designed as a distributed system based on a microservices architecture, supporting modularity, scalability and the independent management of application components.

This architectural choice enables horizontal and autonomous scaling of the various system components, allowing the platform to handle a large number of users concurrently. Furthermore, the distributed architecture ensures reliable data persistence, availability and openness, thereby facilitating future extensions and adaptation to evolving requirements.

Overall, the project represents a practical application of distributed systems principles within the context of a realistic web-based platform.

Contents

1 Goals of the Project

The main goals of the project are:

1. **Create a digital platform for social events**: provide a platform for exploring, discovering social events and interacting with other users.
2. **Support both public access and extended functionalities for registered users**: allow public browsing of events while offering additional functionalities for registered users and organizations.
3. **Promote interaction between users and organizations**: enable users and organizations to communicate and collaborate on the platform.
4. **Increase visibility and credibility of organizations**: enhance organizations' reputation through active community engagement.
5. **Ensure system availability and scalability**: design the platform to handle distributed operations efficiently while maintaining responsiveness.
6. **Guarantee transparency and coherence in the distributed architecture**: ensure users perceive the platform as a reliable and consistent system, even if it is implemented as a distributed architecture.
7. **Enable future extensibility and adaptability**: design the system to easily incorporate new features, services or integrations without major changes.

1.1 Usage Scenarios

This section illustrates the typical interactions between the target users of the platform and the system itself. EvenToNight has been conceived for three types of users: unregistered users, registered users and users registered as organizations.

The use case diagram in fig. 1 summarizes the main interactions of these three user types with the platform.

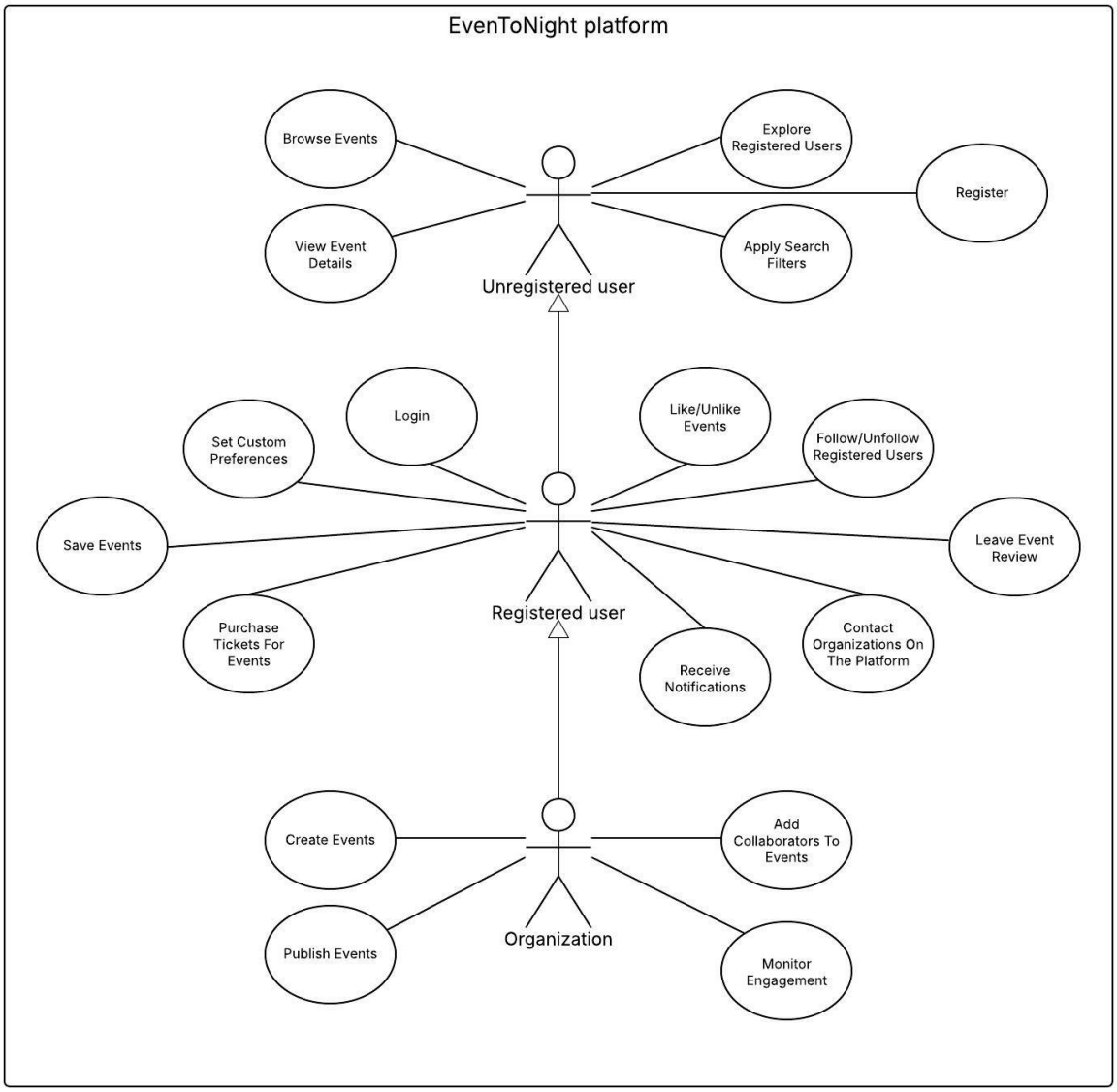

Figure 1: Use case diagram

A practical view of how each type of user interacts with the system according to their needs is provided in the following usage scenarios.

Usage scenario: Browsing and filtering events, exploring other users as an unregistered user.

- **Actor**: Unregistered user.
- **Objective**: Explore events according to location and gather information without registering an account.
- **Main flow**:
  1. The user browses the list of upcoming events.
  2. The user views event details, including poster, description, time and location.
  3. The user applies filters to find events nearby.

4. The user searches for users registered on the platform.

- **Outcome**:
  1. The user accesses all event information without providing personal data.
  2. The user identifies events aligned with their location.
  3. The user discovers other users of the platform.

Usage scenario: Personalized event discovery, ticket purchase and post-event review as a registered user.

- **Actor**: Registered user.
- **Objective**: Discover events based on personal preferences, evaluate them and purchase tickets on the platform.
- **Main flow**:
  1. The user logs in to the EvenToNight platform.
  2. The user navigates their personalized event feed, based on specified interests.
  3. The user selects an event and reads reviews of past events on the organization's page.
  4. The user purchases tickets for the event on the platform.
- **Outcome**:
  1. The user quickly finds events that match their interests.
  2. The user evaluates the organization's reliability and the type of experiences offered.
  3. The purchased ticket appears on the user's personal page and can be downloaded.
- **Post-event action**: After attending the event, the user leaves a review for the event on the organization's page, providing feedback on their experience.

Usage scenario: Browsing the platform with personalized account settings, saving events and contacting organizations as a registered user.

- **Actor**: Registered user.
- **Objective**: Navigate the platform, save events and contact organizations via an interface customized with language and appearance preferences stored in the user account.
- **Main flow**:
  1. The user logs in to the EvenToNight platform.
  2. The user navigates the platform, displayed according to the user's saved language and appearance settings (dark or light mode).

3. The user selects an event from the catalog and saves it to their personal list.
4. The user contacts the organization to request additional information.

- **Outcome**:

1. The user browses events in their preferred language and interface mode.
2. Selected events are stored in the user's personal list.
3. The user can easily communicate with the organization through the platform for support.

Usage scenario: Publication of events and monitoring engagement as an organization.

- **Actor**: Organization.
- **Objective**: Publish events and monitor user engagement.
- **Main flow**:

1. The user logs in to the EvenToNight platform.
2. The user creates an event as a draft.
3. The user adds a collaborating organization to the event.
4. The user publishes the event.

- **Outcome**:

1. The event is successfully published and visible to platform users.
2. The organization receives notifications about new followers and event likes, enabling monitoring of user engagement.

1.2 Definition of Done

The project is considered *done* when the following criteria are met:

1. Requirements fulfilled.

- All main functionalities of the platform, as defined in the functional requirements section, are implemented.
- All business and non-functional requirements specified for the project are satisfied.

2. Testing completed.

- Unit and integration tests for all services pass successfully.
- All API endpoints have been verified using Swagger/OpenAPI.

3. Deploy ready.

- All services are containerized and managed via Docker Compose.
- The simulated distributed environment demonstrates service isolation and internal networking.

- All services can be deployed and executed without errors, ensuring a stable runtime environment.

4. **Documentation updated.**
   - API documentation, architectural diagrams and project documentation are complete, consistent and up to date with the implemented system.

2 Background and Link to the Theory

In the following are presented the main distributed system concepts related to this project, main architectural/interactions patterns and technologies used.

2.1 Distributed Systems Fundamentals

The following core concepts from distributed systems theory are relevant to the design decisions made throughout this project.

- **CAP theorem** states that a distributed system can guarantee at most two of three properties simultaneously: *Consistency* (all nodes see the same data at the same time), *Availability* (every request receives a response) and *Partition tolerance* (the system continues operating despite network partitions). Since partition tolerance is non-negotiable in a real network, systems must choose between consistency and availability under a partition. Systems that prioritise availability accept that replicas may temporarily diverge, relying instead on *eventual consistency*.
- **Eventual consistency** is a consistency model in which, in the absence of new updates, all replicas of a data item will converge to the same value over time. It trades immediate cross-service consistency for higher availability and lower coupling, accepting that different parts of the system may temporarily observe slightly different states.
- **FLP impossibility result** (Fischer, Lynch, Paterson) proves that in a fully asynchronous distributed system, consensus cannot be guaranteed if even a single process may fail. In practice this means failure detection is impossible without some notion of time: a slow process is indistinguishable from a crashed one. **Timeouts** are the practical solution (a process is declared failed if it does not respond within a bounded time) and motivate **exponential backoff** in reconnection logic, spacing out retries to avoid overwhelming a recovering dependency.
- **Availability** is the degree to which a system is operational and responsive when needed. **High availability (HA)** extends this by designing the system to minimise downtime through redundancy and automatic failover: components are replicated across multiple nodes so that the failure of any single node does not interrupt service.
- **At-least-once delivery** is a message delivery guarantee where the broker ensures a message is delivered to the consumer at least once, but may deliver it more than once in the event of failures or retries. Consumers must therefore be designed to tolerate duplicate messages.
- **Idempotency** is the property of an operation whereby applying it multiple times produces the same result as applying it once. In distributed systems, idempotent message handlers are the standard mitigation for at-least-once delivery semantics.

2.2 Architectural Styles

The following architectural styles shape the overall structure and organisation of the system.

- **Microservices** decompose an application into small, independently deployable services, each responsible for a single bounded context. Combined with the **database-per-service** pattern, each service owns its data store exclusively, preventing coupling at the persistence layer and allowing schemas to evolve independently.
- **Event-Driven Architecture** decouples producers and consumers of events, avoiding direct synchronous calls. This reduces temporal coupling, improves scalability and allows new consumers to be added without changing existing producers.
- **Domain-Driven Design (DDD)** provides the modeling vocabulary: a domain is partitioned into *bounded contexts*, each with its own ubiquitous language. Within a context, *aggregates* enforce business invariants, *value objects* represent immutable concepts and *domain events* describe state changes that other contexts may react to.
- **Clean Architecture** organises a service's codebase into concentric layers (domain, application, infrastructure, presentation) with a strict inward dependency rule: outer layers depend on inner ones, never the reverse. This keeps domain logic framework-agnostic and independently testable.
- **CQRS (Command Query Responsibility Segregation)** separates write operations (commands) from read operations (queries) at the application layer, assigning each to a dedicated handler. This makes individual operations smaller and more focused and opens the door to independent scaling of read and write paths.

2.3 Interaction Patterns

The following interaction patterns are used for communication between services and clients.

- **REST over HTTP** is the standard synchronous request/response pattern: the client sends a request and blocks until the server replies. Suitable for operations that require an immediate result.
- **AMQP (message broker)** enables asynchronous, decoupled communication. A producer publishes a message to an exchange; the broker routes it to bound queues; consumers process it independently. A *topic exchange* routes messages by routing key pattern, allowing fine-grained subscriptions. *Quorum queues* replicate messages across cluster nodes for durability and availability.
- **WebSocket** provides a persistent, full-duplex channel between client and server, enabling server-initiated push without polling.
- **Webhooks** allow external systems to push events into the application via HTTP callbacks, triggered by events on the provider's side.
- **Transactional outbox** ensures reliable event publishing: the domain event is written to an outbox table in the same database transaction as the state change; a relay process reads from the outbox and forwards messages to the broker, retrying until success. This decouples message publishing from broker availability.

- **Saga** coordinates a multi-step operation across services without a distributed transaction. Each step has a compensating action that can undo its effect if a later step fails.

2.4 Software Frameworks and Technologies

Technology	**Role**
Docker / Compose / Swarm	Containerisation and orchestration
Traefik	API gateway, reverse proxy, load balancer, rate limiter
Cloudflare + cloudflared	DNS, TLS termination, encrypted inbound tunnel
RabbitMQ	AMQP message broker
MongoDB	Document store (per-service)
PostgreSQL	Relational store (Keycloak)
MinIO	S3-compatible object storage
Keycloak	Identity provider, JWT issuance
Stripe	Third-party payment provider
NestJS	Node.js framework (structured modules, DI)
Node.js + Express	Minimal HTTP server runtime
Scala 3 + Cask	Functional backend services
Vue 3 + Vite	Single-page application frontend
Socket.IO	WebSocket abstraction with Redis pub/sub adapter

Table 1: Software frameworks and technologies

3 Requirements Analysis

3.1 Requirements List

The main requirements that the application must meet are listed below.

3.1.1 Business Requirements

- The platform allows organizations to create and publish posts related to the events they promote.
- The platform allows users to discover events based on their location and interests.
- The platform enables online ticket sales for events, allowing organizations to monetize them.

3.1.2 Functional Requirements

Types of users supported by the system:

- **Unregistered users**, who can explore the platform.
- **Registered users**, who can access features reserved for authenticated users.
- **Users registered as an organization**, who can access all platform features and additionally create and publish events and manage ticketing.

Functional requirements for all users:

- View the Home screen, including interaction modes such as searching for events, viewing popular events, upcoming events and latest additions.
- View all events published on the platform and all registered users from the Explore screen, applying search filters.

Functional requirements for registered users:

- Receive a personalized event feed based on specified interests.
- Like and unlike events.
- Follow and unfollow other registered users.
- Purchase tickets for events.
- Leave a review after attending an event.
- Contact organizations directly within the platform to request support.
- Receive notifications about:
  - new followers
  - new events published by followed organizations
  - new messages

Functional requirements for users registered as an organization:

- Create events, choosing whether to make them public or save them as drafts.
- Specify collaborators when creating events.
- Receive notifications about likes and reviews on their own events.

3.1.3 Non-Functional Requirements

- **Availability**: every request sent to a non-failing node in the system must receive a response.
- **Reliability**: the system must operate correctly over time, minimizing the occurrence of faults.
- **Fault Tolerance**: the system must tolerate failures of individual services (i.e. containerized microservices) without affecting the overall platform operation.
- **Security**: the system requires user authentication and enforces authorization to access resources according to defined rules. In addition, to ensure password confidentiality, only a secure hash of each user's password will be stored.
- **Robustness**: the system must handle incorrect inputs, providing consistent error messages without compromising system stability.
- **Scalability**: the system must be able to handle increasing loads by adding new resources, without degrading performance.
- **Extensibility**: the system must support easy customization and the addition of new functionalities.
- **Maintainability**: the system must be easy to maintain with respect to the addition of new features and system updates. This is supported by well-structured and well-documented code.
- **Accessibility**: the system's graphical interface must be accessible, supporting standard accessibility guidelines.
- **Portability**: the system must be responsive and adapt to different screen sizes and devices, including PCs, tablets and smartphones.
- **Deployability**: the system must automatically update to the latest release version.
- **Architectural Constraint**: the system must be developed following a microservices architecture.

3.2 Top-Down Analysis

3.2.1 Architectural Styles

Starting from the identified requirements, an **event-based architecture** was selected to design the distributed platform.

This architectural style was chosen because it addresses several non-functional requirements of the system. In particular, the following architectural properties directly contribute to satisfying the system's non-functional requirements:

- **Referential and spatial decoupling**: producers and consumers do not reference or depend on each other's location. This supports extensibility, allowing new services to be added without modifying existing components.
- **Temporal decoupling**: events are stored in a persistent event space until consumers are able to process them. This ensures that producer services remain responsive even if consumer services are temporarily unavailable, supporting fault tolerance and availability. In addition, this storage smooths load peaks by decoupling producer and consumer execution timing and it supports extensibility and horizontal scalability by allowing the addition of multiple consumer instances.
- **Loosely coupled services**: producers publish events without waiting for consumers to respond and consumers process events independently. This independence supports fault tolerance, maintainability and extensibility, as temporary service failures or modifications do not affect other components and the addition of new functionalities and service updates is simplified.

At the system boundary level, the platform also follows a **layered architecture**:

- **Presentation layer**: the containerized frontend microservice, providing the user interface.
- **Application layer**: containerized backend microservices exposing REST APIs.
- **Data layer**: repositories interacting with databases for persistence.

The layered organization provides a high-level view of control flow in the distributed platform, highlighting the separation of concerns between user interface, business logic and data management.

4 Design

The system has been designed following a microservices architectural style, where each service models a specific subdomain and exclusively owns its data.

The principal objective of the design phase has been to:

- ensure service autonomy.
- separate services persistence management.
- guarantee scalability and fault isolation.
- preserve local consistency while enabling eventual consistency across distributed services.
- separate responsibilities following domain partitioning.

A key design principle adopted is data ownership per service: no service directly accesses the database of another service.

All inter-service coordination occurs either synchronously or asynchronously. This separation allows the system to tolerate partial failures, avoiding where possible cross-services synchronous operations chaining and maintain loose coupling among components.

4.1 Structure

The identification of domain entities and service boundaries has been guided by principles inspired by Domain-Driven Design (DDD).

The domain has been decomposed into bounded contexts, each representing a coherent business subdomain with its own model and behavior.

This approach has been adopted to:

- reduce semantic ambiguity
- support independence between different subsystems

Each bounded context corresponds to a logically autonomous service, encapsulating:

- domain entities
- business rules
- persistence layer

This alignment between domain boundaries and service boundaries reduces coupling and minimizes the need for distributed transactions, favoring asynchronous coordination and eventual consistency.

Given that, six bounded contexts are identified: user, event, interaction, chat, ticketing and notification.

Moreover, some other services are used: auth (Keycloak), media, payments (Stripe).

4.1.1 Core Entities

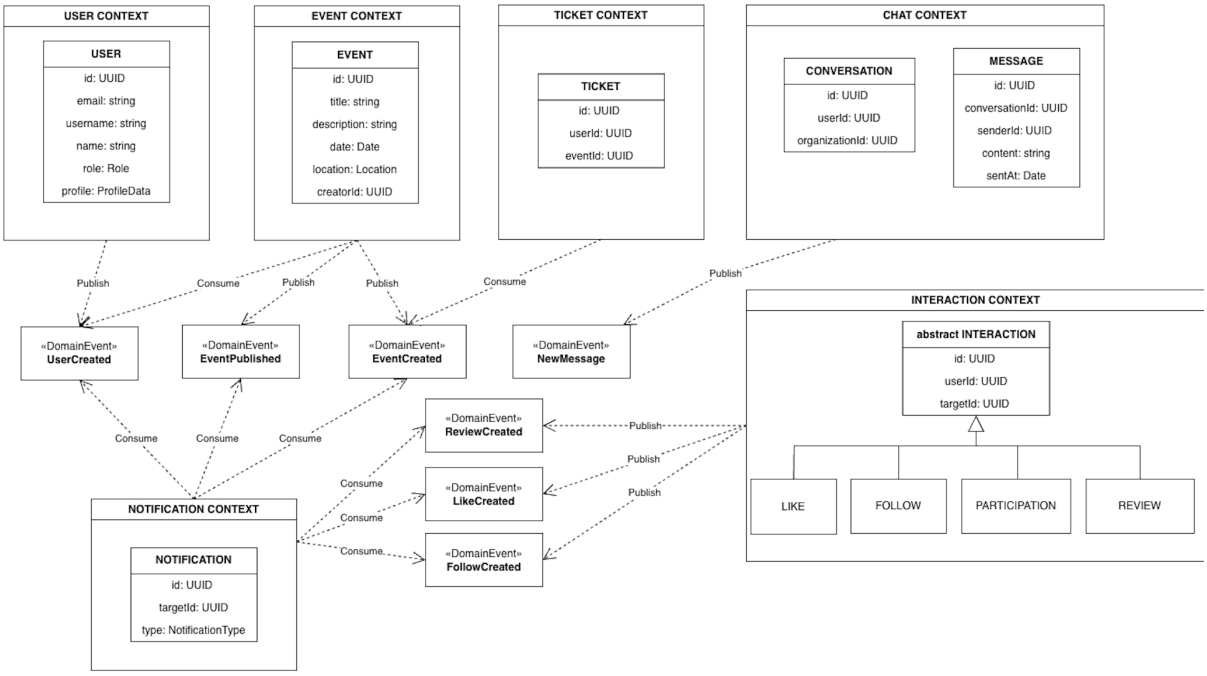

Figure 2: Core entities and bounded context diagram

The diagram shows a simplification of the design, not every aspect of the model is represented in details.

The diagram highlights the separation between bounded contexts and the absence of direct structural dependencies across services.

Cross-service relationships are expressed through other entities identifier and domain events rather than object references or shared persistence.

User Represents a platform account. A User can assume one of two roles:

- Member
- Organization

Attributes include identifier, authentication-related data (account), profile information and events preferences.

Event Represents an activity created by an Organization.

Its attributes include identifier, title, description and all informations about event (e.g. date, time, location).

Ticket Represents a User's receipt of payment to participate in a Event.

It has its own lifecycle (creation, validation or invalidation) and is modeled independently from Event to avoid coupling financial logic with event management logic.

Interaction Abstract concept representing user actions.

In particular:

- Like (User → Event)
- Follow (User → User)
- Review (User → Event)
- Participation (User → Event)

Interactions are separated from Event and User entities to enable event-driven propagation of user activities, allowing them to evolve independently from core domain entities.

Conversation Represents a communication channel between two Users. This entity stores information about the participants involved in the chat and metadata related to the conversation.

Messages are modeled as a separate entity rather than being embedded directly within the Conversation entity. This separation prevents unbounded data growth inside the conversation and allows message data to be managed independently.

Moreover, this design improves efficiency when retrieving the list of conversations. When a user requests the list of active conversations, the system only needs to access conversation metadata without loading all the associated messages. Message data is retrieved only when a specific conversation is opened.

Notification Represents a system-generated alert that must be sent to specific user.

Examples:

- new follower
- new event published from followed organization
- new message/like/review received

Notifications are derived entities: they originate from domain events generated by other services.

4.1.2 Domain Events

In addition to static entities, the system models domain events to improve communication between services.

Examples include:

- `UserCreated`
- `EventCreated`
- `EventPublished`
- `TicketPurchased`
- `LikeAdded`
- `NewMessage`

Domain events enable asynchronous integration between bounded contexts and allow the system to maintain eventual consistency.

4.2 Interaction

Communication between the different components of the system follows two interaction patterns: synchronous request–response and asynchronous event-based communication.

The choice of combining these two approaches allows the system to support direct user interactions while maintaining loose coupling and independent evolution of backend services.

- **Synchronous interactions between frontend and backend services**: Synchronous communication is primarily used for interactions initiated by users through the frontend. In this pattern, a client sends a request to a service and waits for a response before continuing the execution.

  This approach is suitable for operations that require an immediate result, such as authentication, event browsing, or ticket purchasing. The response returned by the service confirms the successful execution of the operation or reports possible errors. Using synchronous interactions for user-driven operations simplifies the control flow and ensures that the user receives immediate feedback from the system.

- **Asynchronous event-driven communication**: Communication between backend services is primarily handled through asynchronous event-driven interactions. In this model, when a service completes an operation that may affect other parts of the system, it emits a domain event describing the change that occurred. Other services can subscribe to the events they are interested in and react accordingly. For example, the creation of an interaction or the publication of an event may trigger the generation of notifications or updates in other subsystems.

Consistency and coordination. Each service guarantees consistency within its own boundaries by executing state-changing operations within local transactions. After a transaction completes, the service may emit one or more domain events describing the resulting state change.

Other services process these events asynchronously, updating their own state or triggering additional operations. This approach avoids the need for distributed transactions across services and allows the system to maintain overall coherence through eventual consistency.

4.3 Behaviour

All of the services in the system follow a common behavioral structure when handling both user requests and inter-service communication. The goal of this design is to maintain strong consistency within each service while enabling coordination between services in a distributed environment.

Three main behavioral patterns can be identified: request-driven operations, event-driven processing and real-time notification propagation.

4.3.1 Request-Driven Operations

User actions are processed through a request-driven workflow. A request is received by the API interface of the corresponding service and forwarded to the layer responsible for executing the domain logic.

To guarantee consistency, operations that modify the system state are executed within a local transactional boundary.

To guarantee reliability, it's adopted the **Outbox Pattern**. Instead of publishing events directly after the state update, domain events are first stored in a dedicated outbox structure within the same transaction as the database update. This ensures that the state change and the corresponding domain event recording occur atomically. Once the transaction successfully completes, the events stored in the outbox can be asynchronously delivered to the message broker.

The typical workflow can therefore be summarized as follows:

1. A client request is received through the service API.
2. The request is validated and processed (optionally, also making synchronous request to other services) by the service logic.
3. A local transaction updates the service state and if necessary records in the outbox the domain events describing the change.
4. After the transaction completes, the events are asynchronously published.

This pattern guarantees internal consistency within the service while enabling reliable propagation of domain events to the rest of the system.

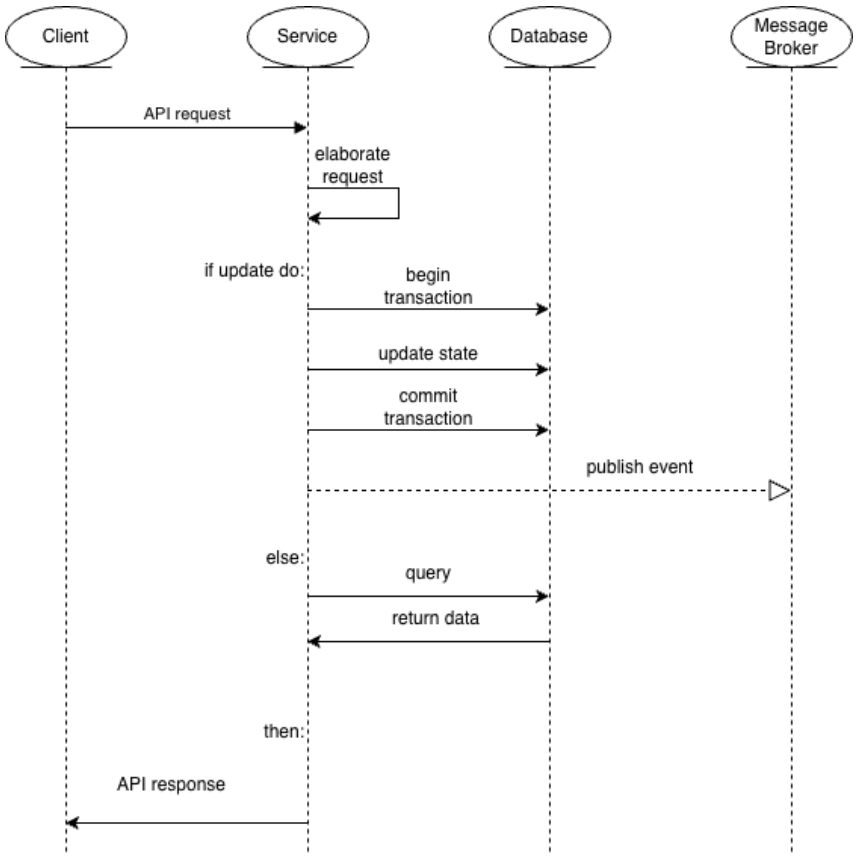

Figure 3: Request-driven behavior

4.3.2 Event-Driven Processing

Services also react to domain events generated by other services. When an event is received, the service processes it through its domain logic and may update its internal state or trigger additional events.

This approach enables coordination between bounded contexts without requiring direct dependencies between services.

The typical flow for this interaction is:

1. A domain event is received.
2. The service processes the event through its domain logic.
3. If required, a local transaction updates the service state.
4. Additional domain events may be generated.

This event-driven approach allows services to collaborate asynchronously while preserving loose coupling and independent evolution.

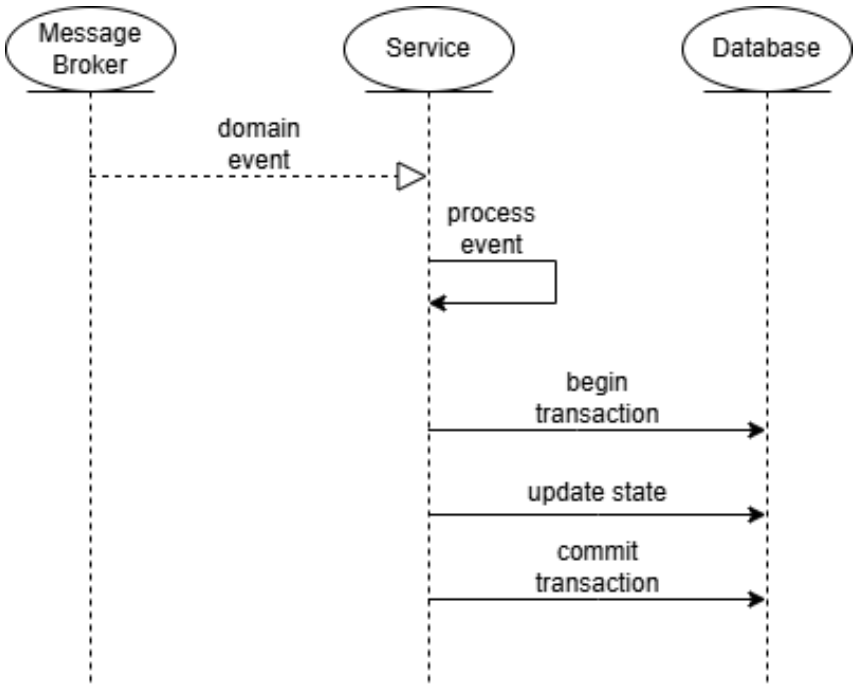

Figure 4: Event-driven behavior

4.3.3 Real-Time Notification Propagation

A specific behavior is implemented for real-time user notifications. Some domain events represent user actions that must be sent to other users, such as a message or a new interaction. These events trigger the creation of notification entities.

The notification subsystem processes the relevant events, stores the corresponding notification and propagates it to connected clients through a real-time communication channel.

The workflow is the following:

1. A domain event is generated by a service.

2. The notification subsystem consumes the event.

3. A notification entity is created and stored.

4. The notification is delivered to relevant connected clients in real-time.

This mechanism allows the system to react promptly to user-relevant events while maintaining a decoupled architecture.

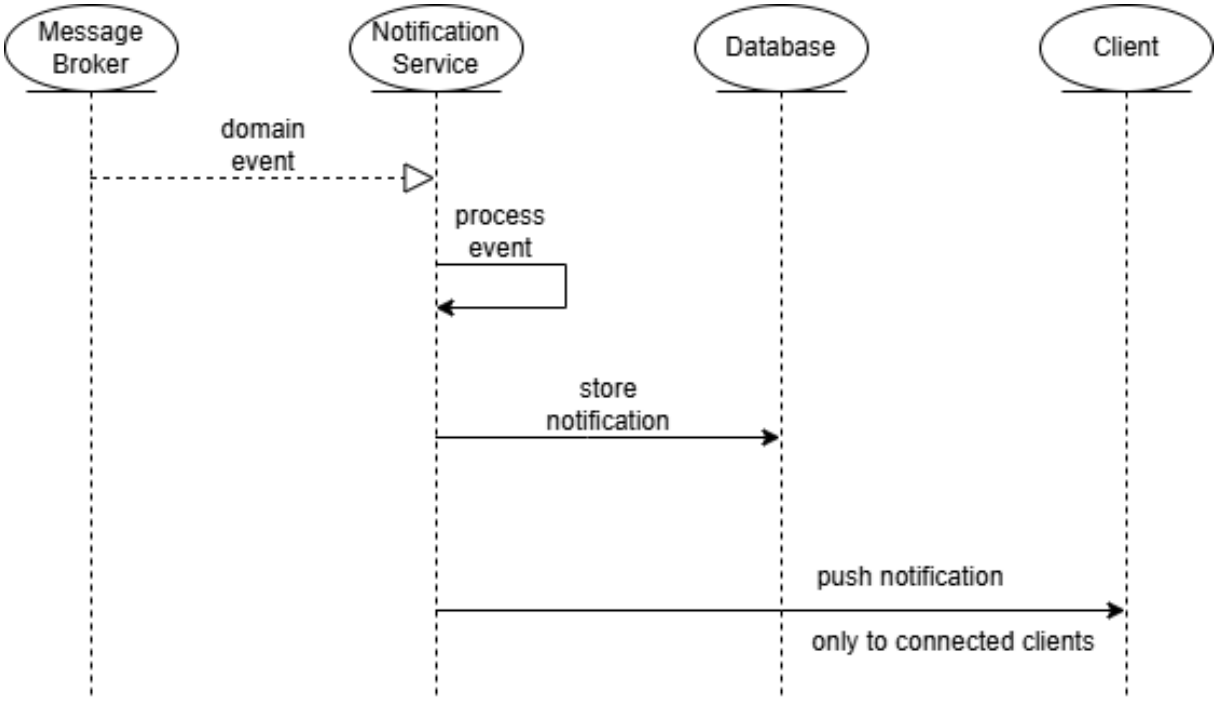

Figure 5: Real-time notification propagation behavior

4.4 Architecture

4.4.1 Architectural Style

The system is designed according to a **microservices architecture**. The application is decomposed into a set of small, independently deployable services, each responsible for a specific bounded context: user, event, interaction, chat, ticketing and notification.

Each microservice encapsulates its own domain logic and adopts the **database-per-service** pattern.

When shared information is required (e.g., basic user data such as `id`, `name` and `avatar` needed by the `chat` or `interaction` services), it is propagated via **domain events** rather than retrieved through synchronous queries. This intentional data duplication reduces communication overhead and keeps services decoupled.

Moreover, this design avoids the bottleneck that would arise from sharing a single database across multiple services and guarantees strong data ownership by preventing services from directly accessing another service's data store.

Inter-service communication

Two complementary communication styles are adopted:

- **Synchronous (HTTP REST)**: used when fulfilling a request requires invoking an operation or retrieving data from another service, or to propagate state changes immediately to dependent services. For example, when an event is created, the `event` service notifies `ticketing` via its internal API so that ticket-related operations can proceed right away.
- **Asynchronous**: used to propagate state changes across service boundaries without creating runtime dependencies between them and to push real-time updates to connected clients. Three mechanisms are in place:
  - **Domain events (RabbitMQ)**: services publish events to the message broker; subscribers consume them independently. For example, when an event is published, the `event` service emits an `EventPublished` domain event consumed by the `notification`, `ticketing` and `interaction` services.
  - **WebSocket (Socket.IO)**: used by the `notification` service to maintain persistent connections with clients, enabling server-side push of real-time updates without polling. For example, when `notification` receives an `EventPublished` domain event, it delivers an update to the relevant connected clients. In this case, users following the organization that published the event.
  - **Webhooks**: used for third-party integrations that push notifications into the system, for example payment provider (Stripe) callbacks delivered to the `ticketing` service. The receiving service processes the incoming HTTP call asynchronously without blocking any client-facing request.

API gateway

All external traffic flows through an API gateway implemented with **Traefik**, which acts as a single entry point and keeps the internal service topology invisible to clients.

Motivation for the architectural choice

- **Service independence**: each service can be developed, tested and deployed independently.
- **Scalability**: services can scale horizontally according to their specific load.
- **Loose coupling**: the combination of database-per-service and event-driven communication minimises direct dependencies, improving maintainability.
- **Data isolation**: each service's database schema evolves independently, preventing unintended cross-service coupling.
- **Resilience and fault isolation**: failures are contained within individual services; asynchronous communication through the message broker allows a service to be temporarily unavailable without blocking the rest of the system.

4.4.2 Infrastructure

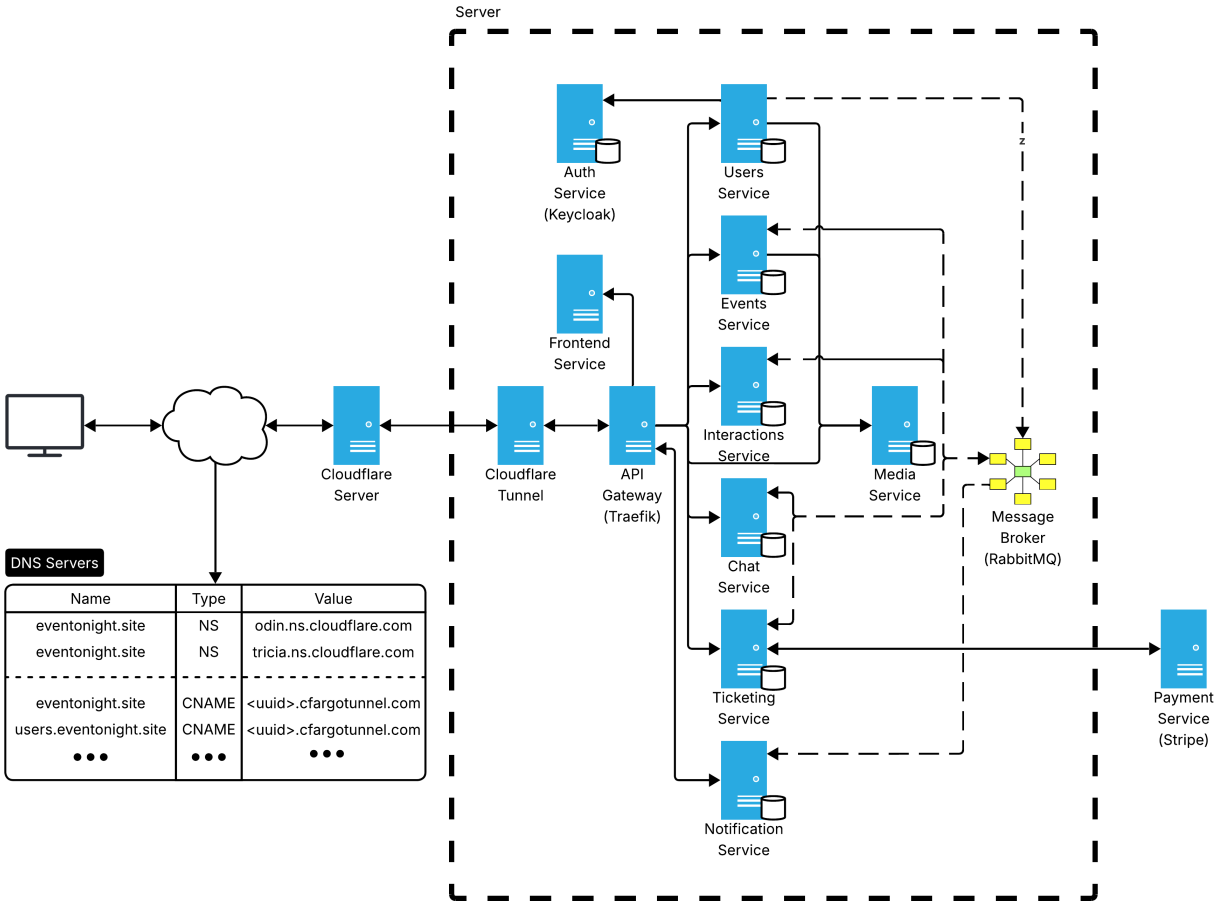

Figure 6: Infrastructure diagram

Components

All infrastructural components are listed in the table below.

Component	Technology	Description
DNS / TLS	Cloudflare	Authoritative DNS and TLS termination
Tunnel daemon	cloudflared	Maintains an outbound encrypted tunnel to Cloudflare; no inbound ports required on the cluster
API gateway / Reverse proxy	Traefik v3	Single entry point for all external HTTP/WebSocket traffic; handles routing, load balancing, rate limiting, retries, circuit breaking and request timeouts; discovers services via Docker label-based configuration
Backend services	Scala 3 + Cask	`user`, `event`
Backend services	NestJS (Node.js)	`chat`, `interaction`, `ticketing`
Backend services	Express (Node.js)	`media`, `notification`
Frontend	Vue 3 + Vite	Single-page application served as static assets
Identity provider	Keycloak	User authentication and JWT issuance
Message broker	RabbitMQ	Asynchronous event delivery via a topic exchange
Document databases	MongoDB	One dedicated instance per service
Relational database	PostgreSQL	Backing store for Keycloak
Object storage	MinIO	Binary file and image storage for the `media` service
Payment provider	Stripe *(external)*	Third-party payment processing
Seed service	Node.js	Populates databases with initial data on first deployment

Table 2: Infrastructure components

Network distribution

With the exception of Stripe, all components run inside a single host using **Docker Compose**. No service port is exposed to the public internet, the only inbound path is through the Cloudflare tunnel.

Two categories of Docker bridge networks are used:

- **`eventonight-network`**: shared by all application services, Traefik, RabbitMQ and Keycloak. Used for inter-service communication and Traefik routing.
- **`<service>-network`**: one private network per service, shared exclusively between the service, its database instance and any auxiliary containers (e.g. provisioning scripts that perform initial setup, as in the case of Keycloak). This enforces database isolation, no other service can reach another service's database.

Service discovery

External clients resolve subdomains through Cloudflare's authoritative DNS. Each subdomain is a CNAME record pointing to `<uuid>.cfargotunnel.com`, which resolves to a Cloudflare edge IP. Traffic enters the cluster through the `cloudflared` tunnel and is handed off to Traefik.

Inside the host, discovery is handled by Docker Compose:

- **Internal DNS**: Compose registers each service by name in the internal DNS (e.g., `http://users:9000`, `amqp://rabbitmq:5672`). No manual registration is required.
- **Traefik routing**: Traefik watches the Docker API for label changes and rebuilds its routing table automatically when services are started or stopped.

4.4.3 Data Flow

Synchronous request flow (HTTP REST)

```
Browser → Cloudflare (DNS + TLS) → cloudflared
        → Traefik → Service → Database → response
```

The browser sends an HTTPS request to a subdomain (e.g., `events.eventonight.site`). Cloudflare resolves the DNS and terminates TLS, then injects the request into the persistent encrypted tunnel (QUIC) maintained by `cloudflared` on the cluster. Traefik receives it, matches the `Host` header against its routing rules and forwards the request to the target service. The service handles the request, queries its own database instance and returns the response through the same chain.

Asynchronous event flow (AMQP)

```
Service → Exchange (eventonight) → Queue (per-service) → Consumer service
```

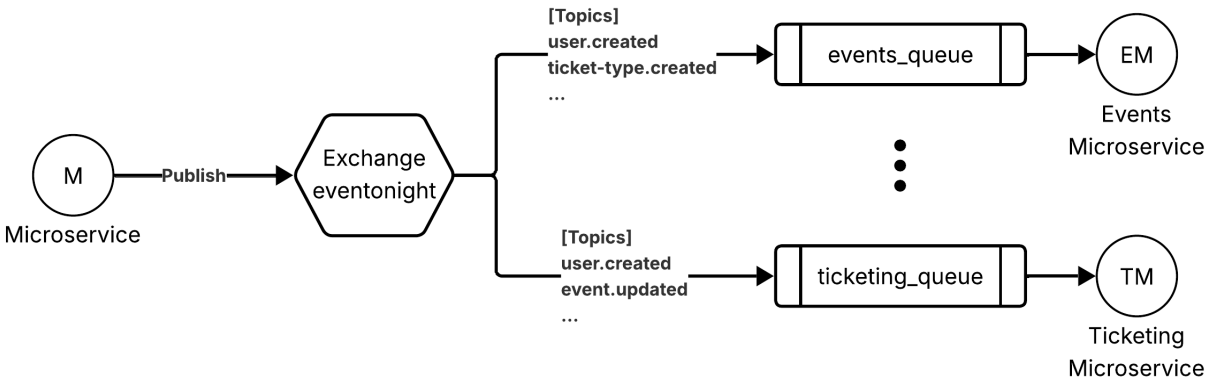

Figure 7: Message broker topology

RabbitMQ uses a single **topic exchange** named `eventonight`. Each consuming service has a dedicated queue bound to the exchange with a set of topic subscriptions (e.g., `ticketing_queue` binds to `user.created`, `event.updated` and many others). When a service publishes a domain event with a routing key, the exchange fans it out to every queue whose binding pattern matches, without the publisher knowing who the consumers are.

Real-time push (WebSocket)

```
notification service → Socket.IO → Browser
```

The `notification` service consumes events from its queue and pushes real-time updates to the relevant connected clients over WebSocket (Socket.IO).

Webhook flow (Stripe → ticketing)

```
Stripe → Cloudflare → cloudflared → Traefik
       → ticketing (acknowledged immediately, processed asynchronously)
```

Stripe delivers payment event callbacks via HTTP POST to the `ticketing` service. The service responds immediately with `200 OK` and processes the event asynchronously, without blocking any client-facing operation.

4.4.4 Web API

All application services expose a **REST API**, accessible externally via Traefik at `<service>.eventonight.site`. Every protected endpoint requires a **JWT Bearer token** in the `Authorization` header, issued by Keycloak and validated independently by each service using Keycloak's public key, no centralised auth gateway is needed. The `notification` service additionally exposes a **WebSocket endpoint**.

4.4.5 Scalability and High Availability

The current deployment runs all components on a **single host** using Docker Compose. Each service runs as a single container instance, which provides basic availability but represents a single point of failure at the host level. The architecture below is designed to transition to **high availability** by adopting **Docker Swarm**.

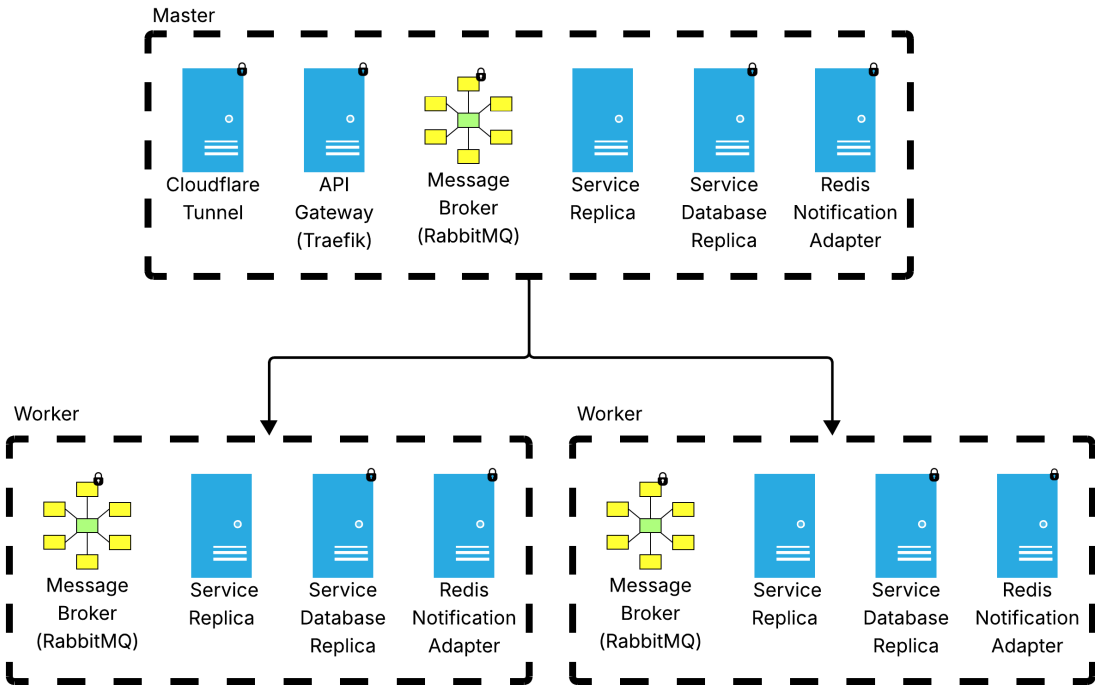

Figure 8: Docker Swarm deployment topology

Even on a single node, Swarm already provides a first step toward HA through **rolling updates**, services can be updated replica by replica with zero downtime, unlike a plain Compose deployment where an update implies a brief interruption. Moving to a multi-node cluster then enables true HA by distributing replicas across machines, so that the failure of a node does not take down the service.

However, adding nodes and replicas is a necessary but not sufficient condition for HA, it also requires deliberate design choices at the infrastructure level and at the application level to preserve correctness: stateful services require dedicated replication strategies and deploying multiple replicas of a service introduces competing consumer scenarios on the broker that must be explicitly handled.

Cluster topology

For true HA, the Swarm should have at least **three manager nodes**. Swarm managers use the Raft consensus protocol to maintain cluster state and an odd number of managers is required to form a quorum and tolerate node failures (three managers tolerate one failure). Each manager node should run its own Traefik and `cloudflared` instance, so Cloudflare load-balances incoming traffic across all managers automatically.

Service placement

Stateless application services can be freely distributed across nodes. Stateful services (RabbitMQ and databases) require **placement constraints** to be distributed and binded on different machines. Manager nodes should be set to **drain** availability so the Swarm scheduler assigns no application workloads to them, keeping them dedicated exclusively to cluster management and incoming traffic.

Message broker with multiple consumers

Scaling application services to multiple replicas introduces a non-trivial consideration for broker consumption. The current setup uses a **single active consumer** per queue: only one replica consumes from a given queue at a time, guaranteeing FIFO message processing. This is sufficient for HA because if the active consumer fails, another replica takes over, but it does not improve throughput.

To scale throughput beyond a single consumer, more refined strategies can be explored:

- **Consistent hash exchange**: a RabbitMQ plugin that replaces the standard topic exchange with a hashing layer. Using that, messages are routed to one of N queues based on a hash of the routing key, allowing N consumers to process in parallel while preserving per-key ordering.
- **Streams / Super Streams**: RabbitMQ Streams provide a persistent, replayable log. Super Streams partition a logical stream across multiple physical streams, each consumed by a dedicated replica, combining throughput scaling with ordering guarantees per partition.
- **Sequence fields**: adding a monotonically increasing `version` or `seq` field per entity allows consumers to detect and discard out-of-order messages regardless of the consumption strategy adopted.

What changes with Swarm:

- **`cloudflared` and Traefik** scale with the number of manager nodes, one instance per manager. Traefik natively supports Swarm mode via the Docker API.

- **Stateless application services** can be scaled to multiple replicas and distributed freely across nodes.
- **Stateful services** require dedicated HA strategies: MongoDB moves to a **replica set** (primary + secondaries) and RabbitMQ to a **quorum queue cluster**, both require an odd number of members (e.g. 3, 5) to guarantee quorum and avoid split-brain, ensuring data redundancy and automatic failover. Each stateful instance must be pinned to a distinct node (at most one replica per node) so that a single node failure does not take down multiple members of the same cluster. Replicated data stores also improve **fault tolerance**, the system continues to operate and preserves data integrity even if individual nodes are lost.
- **WebSocket sticky sessions**: when `notification` runs as multiple replicas, Traefik must be configured with a sticky cookie to ensure Socket.IO handshake and subsequent frames are consistently routed to the same replica. However, sticky sessions alone are not sufficient: to broadcast events to all connected clients regardless of which replica they are connected to, a **Redis adapter** for Socket.IO is required to propagate messages across all instances.

4.5 Corner Cases

The system is designed to be **highly available** and **eventual consistent**. Each service operates independently and failures are isolated: when a component goes down, only the functionality directly depending on it is affected, while the rest of the system continues to operate.

Fault detection. Each service exposes a `/health` endpoint and declares a `healthcheck` in Docker Compose. This allow Docker to automatically restarts or reschedules tasks that become unhealthy, making fault detection active. To further improve observability, a dedicated monitoring service could be introduced.

Startup ordering and reconnection. In a Compose deployment, dependent services use `depends_on` with `condition: service_healthy` to enforce startup ordering. In a Swarm deployment this mechanism is not available: each service must instead be resilient to the temporary unavailability of its dependencies and implement reconnection logic autonomously. Note that reconnection logic is necessary regardless of the deployment model, as dependencies may also fail and recover during normal operation. All reconnection attempts, to MongoDB, RabbitMQ and other services, use an exponential backoff strategy.

Error signalling. Every API endpoint returns standard HTTP status codes to communicate failures to clients: `400` for validation errors, `401` for missing or invalid tokens, `403` for authorisation failures, `404` for resources not found, `409` for conflicts, `500` for unexpected server errors and so on. With a monitoring service in place, an alerting mechanism can be added to proactively notify on service failures.

Component failure and recovery. In a Compose deployment, all containers run with `restart: unless-stopped`, so Docker automatically restarts any crashed container.

Docker itself is configured to start on host boot, meaning that even a full machine reboot brings the entire system back up without manual intervention. In a Swarm deployment, the restart policy is managed by the Swarm scheduler, which restarts failed tasks and, if a node goes down, reschedules its tasks onto healthy nodes automatically. In both cases, full recovery is achieved because services are resilient to dependency failures and re-establish connections autonomously.

Database failure. If a MongoDB instance goes down, its owning service remains partially operational: requests that do not require the database can still be served. The service reconnects automatically when the database comes back up. In a Swarm deployment with a replica set, a single node failure triggers an automatic primary election and the service reconnects to the new primary, avoiding a full outage. In both cases, multi-document operations are executed within MongoDB transactions, so a crash mid-operation leaves no inconsistent state, the transaction is simply rolled back.

Service failure. If a service crashes, Docker restarts it automatically. Since all writes are transactional, there is no risk of partial state being committed. However when a service crashes after initiating an outbound HTTP call to an external system (e.g. creating a Stripe checkout session or a Keycloak user) but before completing its local operations, an inconsistency may arise: the side effect on the external system has already occurred but the local state does not reflect it. In these cases specific strategy must be applied for each integration to handle it correctly.

RabbitMQ failure. All queues are declared as durable with persistent messages, so any message received by RabbitMQ before it goes down is safely stored to disk and delivered to consumers once it restarts. In a Swarm deployment with a quorum queue cluster, a single node failure does not cause an outage, the cluster continues operating as long as a majority of nodes are available and messages are replicated across them. RabbitMQ provides at-least-once delivery semantics; services handle this by designing message handlers to be idempotent. If RabbitMQ is unavailable at the moment a service needs to publish, messages are not lost since each service implements the outbox pattern.

Keycloak failure. If Keycloak goes down new logins and token refreshes will fail. However, services validate JWTs locally using Keycloak's public key, so requests carrying a still-valid token continue to be accepted until expiry. Once Keycloak restarts, authentication resumes normally.

Traefik or cloudflared failure. In a Compose deployment, external traffic cannot enter the system until Docker restarts the container and access resumes. In a Swarm deployment, both run one instance per manager node, so a single manager failure does not interrupt external traffic, Cloudflare simply routes through the remaining instances.

Ticketing crash during checkout session creation. If `ticketing` crashes after tickets have been reserved and the Stripe session has been created, but before send response to the client, the session will eventually expire and Stripe delivers a `checkout.session.expired` webhook allowing `ticketing` to release the reserved tickets. If the crash occurs before the Stripe session is created, no webhook will ever arrive and the tickets remain stuck

in `PENDING`; a background cleanup job on timed-out pending orders would be needed to handle this case.

Ticketing unavailability. Webhook callbacks from Stripe cannot be processed. Stripe automatically retries undelivered events for up to three days, so the `ticketing` service will receive and process them once it recovers, without any data loss.

Dual notification path (sync + async). When an event is created, the `event` service notifies `ticketing` synchronously via HTTP call to an internal endpoint, without waiting for the asynchronous domain event to arrive. The RabbitMQ message still follows as a fault-tolerance fallback.

User crash during register and password update. If `user` crashes after creating the account in Keycloak but before persisting the user record locally, the user exists in the identity provider but not in the `user` service. On the next login attempt, Keycloak authenticates the user and issues a valid JWT, but `user` has no matching record. A recovery strategy can be to detect this condition at login time and create a minimal empty profile automatically, mirroring what registration would have done. For password updates, a crash mid-operation leaves the state consistent but the user may not have been correctly notified of the outcome; a "forgot password" flow provides a user-friendly self-service recovery path in this case.

Consistent backup. Because each service owns its own database, there is no single point from which to take a globally consistent snapshot. Backing up each database independently means the captured states may be at slightly different points in time. Full cross-service consistency at backup time is not achievable without a distributed snapshot protocol. One practical mitigation is to temporarily halt writes across all services, snapshot all databases simultaneously, then resume, guaranteeing consistency at the cost of a short downtime.

5 Salient Implementation Details

Full API documentation has been produced for all services:

- **OpenAPI specs** (REST)
- **AsyncAPI specs** (RabbitMQ)

The sections below cover implementation choices not captured by the API specifications.

5.1 Technological Details

In-transit data representation. All REST APIs and RabbitMQ messages use **JSON** as the serialization format.

Database querying. All application services use **MongoDB** as their document store:

- `user` and `event` (Scala) query MongoDB via the raw Java driver (`mongodb-driver-sync`).
- `chat`, `interaction`, `notification` and `ticketing` (Typescript) use **Mongoose** as the ODM, with schema decorators and typed model queries.

The `media` service stores binary files in **MinIO**, an S3-compatible self-hosted object store. PostgreSQL is only used internally by Keycloak.

Authentication. Authentication is handled by **Keycloak**. On login, Keycloak issues a short-lived JWT access token and a longer-lived refresh token. Every protected API endpoint requires an `Authorization: Bearer <token>` header. Each service validates the JWT independently using Keycloak's RSA public key, the `user` service exposes a `/public-keys` endpoint that the other services fetch at startup. No centralised auth gateway is involved in validating requests.

The frontend stores tokens in `sessionStorage`, automatically refreshes the refresh token before expiry and injects access token into all outgoing requests.

Authorization. The system uses **Role-Based Access Control (RBAC)** with two roles: `member` and `organization`. Roles are assigned in Keycloak at registration time and embedded in the JWT. Services extract the role from the token and apply authorization logic at the handler level, for example, only an `organization` can create events and only the owning user can modify their own resources.

RabbitMQ message format. All domain events published to the `eventonight` topic exchange share a common envelope:

```
{
  "eventType": "event.published",
  "occurredAt": "2025-01-01T12:00:00Z",
  "payload": { ... }
}
```

Listing 1: Domain event envelope

MongoDB single-node replica set. MongoDB transactions require a replica set. All services run MongoDB in replica set mode even on a single node, using `-replSet rs0` and initialising a one-member replica set at startup. This setup unlocks multi-document transaction support and makes the transition to a multi-node replica set in a Swarm deployment straightforward, with no changes required to the application layer.

Stripe webhook authentication. Stripe signs every webhook payload using a secret. The `ticketing` service verifies this signature before processing any incoming webhook, ensuring that only legitimate Stripe events are accepted.

Socket.IO: authenticated connections and scaling. WebSocket connections to the `notification` service are authenticated: the client sends its JWT and `userId` during the Socket.IO handshake and the server validates the token before allowing the connection. Socket.IO was chosen over raw WebSocket also because it natively supports a Redis pub-/sub adapter, which would allow the `notification` service to scale horizontally across multiple replicas while still delivering pushes to the correct client regardless of which replica it is connected to.

Key implementation patterns.

- **Transactional outbox**: each service writes domain events to an outbox collection atomically with the state change (within a MongoDB transaction); a relay process forwards them to RabbitMQ, retrying with exponential backoff until delivery succeeds. This decouples message publishing from the availability of RabbitMQ.
- **Saga pattern**: the ticket purchase flow is implemented as a two-phase saga. Phase 1 reserves inventory and creates an order atomically in MongoDB (TX1). Phase 2 calls Stripe to create a checkout session. If Stripe fails, a compensating transaction releases the reserved tickets and cancels the order. If the service crashes between the two phases, the compensation does not run, tickets remain in `PENDING` state indefinitely. The correct mitigation would be a scheduled cleanup job that releases orders stuck in `PENDING` beyond a timeout threshold; this is a known limitation of the current implementation.

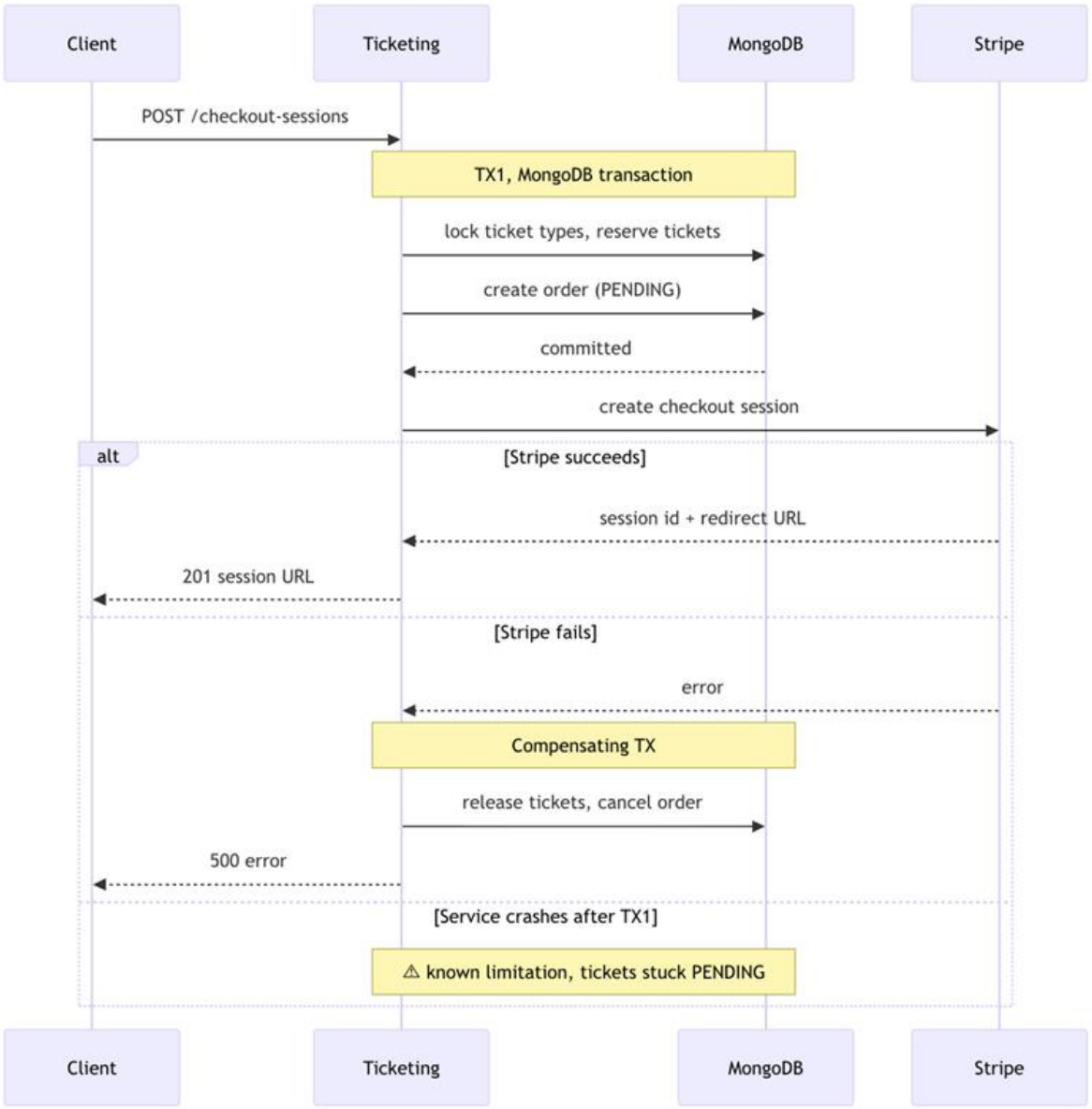

Figure 9: Ticket purchase saga sequence diagram

5.2 Service Architecture Patterns

Services adopt different internal architectural styles.

Clean Architecture (`ticketing`, `user`, `event`, `notification`). Some services are structured around **Clean Architecture** with **Domain-Driven Design** principles. The codebase is organised into four layers with strict inward dependency rules:

- `domain/`: contains aggregates, value objects, domain events, repository interfaces and domain service interfaces. No dependency on any framework or infrastructure.
- `application/`: contains use cases, application services and DTOs. Orchestrates domain logic without touching infrastructure directly.

- `infrastructure/`: contains concrete implementations of repository interfaces (MongoDB), external service adapters (Stripe, Keycloak), mappers between domain objects and persistence schemas.
- `presentation/`: contains HTTP controllers, RabbitMQ consumers and WebSocket gateways. Translates inbound requests into application-layer calls.

This separation ensures that domain logic is fully isolated from infrastructure concerns and it's independently testable. Within this structure, the organization of the application layer varies by service: `event` and `notification` use **Command Query Responsibility Segregation** (CQRS) pattern, with write operations represented as commands and read operations as queries, each handled by a dedicated class. The remaining services use a unified application service or handler approach without explicit read/write separation. The current implementation uses a single shared data model for reads and writes; fully separating the read and write models would unlock further performance and scalability gains and is a natural evolution of the current design.

NestJS module organisation (`chat`, `interaction`). Some services adopt the **NestJS default module structure**: the codebase is organised by feature module (e.g. `conversations`, `users` in `chat`), each encapsulating its own controllers, services and Mongoose schemas.

6 Validation

The system has been validated through a combination of automated testing and API-level verification, with the objective of verifying both the internal correctness of services and the proper behavior of exposed interfaces.
Two main testing approaches have been adopted: unit and integration testing of service logic, and API-level testing through interactive documentation.

6.1 Automated Testing and Coverage

Each service has been tested through automated tests designed to cover:

- domain logic and service-layer behavior;
- validation rules and edge cases.

Integration tests of data persistence operations have been performed using a MongoDB in-memory implementation or by a MongoDB container automatically started (and torn down) during Gradle test execution.
To assess the effectiveness of the test suite, code coverage has been measured. The project achieves an overall coverage of at least 70%, ensuring that the majority of the codebase is exercised during testing. This level of coverage provides reasonable confidence in the correctness of the implemented logic.

6.2 API Testing

In addition to automated tests, the system has been validated through direct interaction with the exposed APIs.
Each service provides a formal API specification compliant with the OpenAPI standard, which allows endpoints to be explored and tested interactively. This interface has been used to manually verify the correctness of request handling, response formats and error management.
API testing has been particularly useful for:

- validating request/response workflows;
- testing edge cases and invalid inputs;
- verifying correctness of response formats.

The full set of APIs is available and can be consulted at the following link: OpenAPI specs (REST).
Moreover, APIs have been tested in integration with the frontend, ensuring correct implementation and usage.

7 Deployment Instructions

The application supports two deployment modes: **Docker Compose** for single-node setups and **Docker Swarm** (beta) for multi-node, highly available deployments. In both cases, all services run as independent containers and are managed via a centralised script.

7.1 Installation

1. Clone the repository.

```
git clone https://github.com/EvenToNight/EvenToNight.git
cd EvenToNight
```

2. Configure environment variables.

```
cp .env.template .env
# Edit .env and fill in all required fields
# Note: if using the --no-deps flag, Stripe keys can contain
   arbitrary values. All fields must still be filled in.
```

7.2 Docker Compose

Option A: Use pre-built images from ghcr.io (Recommended).

```
# Pull images
./scripts/composeApplication.sh pull

# Pull images with database seeding
./scripts/composeApplication.sh --init-db pull

# Deploy the application
./scripts/composeApplication.sh up -d --wait

# Deploy with database seeding
./scripts/composeApplication.sh --init-db up -d --wait

# Deploy in development mode (with host-mapped ports and
   dashboards for databases, RabbitMQ and Traefik)
./scripts/composeApplication.sh --init-db --dev up -d --wait
```

Option B: Local build. Add the `--build` flag to build services locally instead of using pre-built images. The `--dev` flag is required as it includes the build instructions:

```
# Build and deploy
./scripts/composeApplication.sh --dev up --build -d --wait

# Build and deploy with seeding
./scripts/composeApplication.sh --init-db --dev up --build -d --
   wait
```

Additional flags.
`--no-deps`: Excludes external dependencies (Stripe). The flag can be added to any deploy command to exclude external services:

```
# Deploy without external dependencies
./scripts/composeApplication.sh --no-deps up -d --wait

# Deploy with seeding but without external dependencies
./scripts/composeApplication.sh --init-db --no-deps up -d --wait
```

Note: When using `--no-deps`, the Stripe keys (`STRIPE_SECRET_KEY`, `STRIPE_PUBLISHABLE_KEY`, `STRIPE_WEBHOOK_SECRET`) in `.env` can contain arbitrary values.

`--project-name`: Overrides the Docker Compose project name (defaults to `eventonight`):

```
./scripts/composeApplication.sh --project-name myproject up -d --
    wait
```

Stripe configuration. For Stripe payments in a local environment (required only if NOT using `--no-deps`):

```
./services/payments/scripts/local-webhooks.sh
```

This script must be run to forward Stripe webhooks to the local environment. For more information on using sandbox mode, refer to the Stripe documentation.

Alternative setup. Use Gradle to set up the entire environment with seeding and the Stripe listener:

```
./gradlew setupApplicationEnvironment
```

Teardown.

```
./scripts/composeApplication.sh down
./scripts/composeApplication.sh down -v   # also remove volumes
./gradlew teardownApplicationEnvironment  # via Gradle
```

7.3 Docker Swarm (Beta)

Prerequisites.

Initialise the swarm on the manager node:

```
docker swarm init
```

Join additional worker nodes using the token provided by the manager:

```
docker swarm join --token <token> <manager-ip>:2377
```

Option A: Use pre-built images from ghcr.io (Recommended).

```
./scripts/deploySwarm.sh
```

The `--auto-labels` flag can be added to automatically assign placement labels to nodes in a balanced way, without having to configure them manually:

```
./scripts/deploySwarm.sh --auto-labels
```

Note: On the first deploy, database seeding is performed automatically.

Option B: Local build.
Build images locally and push them to Docker Hub (multi-arch) so all workers have access to them, then deploy:

```
./scripts/deploySwarm.sh --local --build
```

To override the hostname baked into the frontend image (defaults to `HOST` from `.env`, e.g. to use `localhost` for local testing):

```
./scripts/deploySwarm.sh --local --build --host <hostname> --auto
   -labels
```

To deploy using already-pushed local images (without rebuilding):

```
./scripts/deploySwarm.sh --local
```

To build and push without deploying:

```
./scripts/deploySwarm.sh --build
```

Additional flags.
`--no-deps`: Same behaviour as in Docker Compose: excludes external dependencies. Stripe configuration applies equally (see p. 36).

```
./scripts/deploySwarm.sh --no-deps
```

`--stack-name`: Overrides the stack name (defaults to `eventonight-swarm`):

```
./scripts/deploySwarm.sh --stack-name mystack
```

Recovery.
If some services are not fully running after a deploy, the recovery script force-updates only the failing ones:

```
./scripts/swarmRecover.sh [STACK_NAME]
```

To check the current status of all services without recovering:

```
./scripts/swarmRecover.sh [STACK_NAME] --status
```

`STACK_NAME` defaults to `eventonight-swarm`.
Teardown.

```
./scripts/deploySwarm.sh --stop
./scripts/deploySwarm.sh --stop --remove-volumes
./scripts/deploySwarm.sh --remove-local-images  # remove test
   images from Docker Hub
```

Alternatively, the application is already running in production at EvenToNight.

8 Usage Examples

Below are some examples of interaction with the application, illustrating the main usage flows.
Specifically, the following use cases are shown:

- **Explore the platform**
- **Login and Registration**
- **Create an event**
- **Attend an event**
- **Review an event**
- **Contact an organization**

The storyboards are presented directly using the developed platform, showing the main user interactions.
In the examples below, some secondary or repetitive navigation dynamics have been omitted.

8.1 Explore the Platform

A user who opens the platform can start exploring it. In particular, they can discover proposed events or search for them directly via the search bar. If they want to see more results, they can go to a dedicated page for exploring the platform content, where it is possible to filter events and more conveniently search for users and organizations.

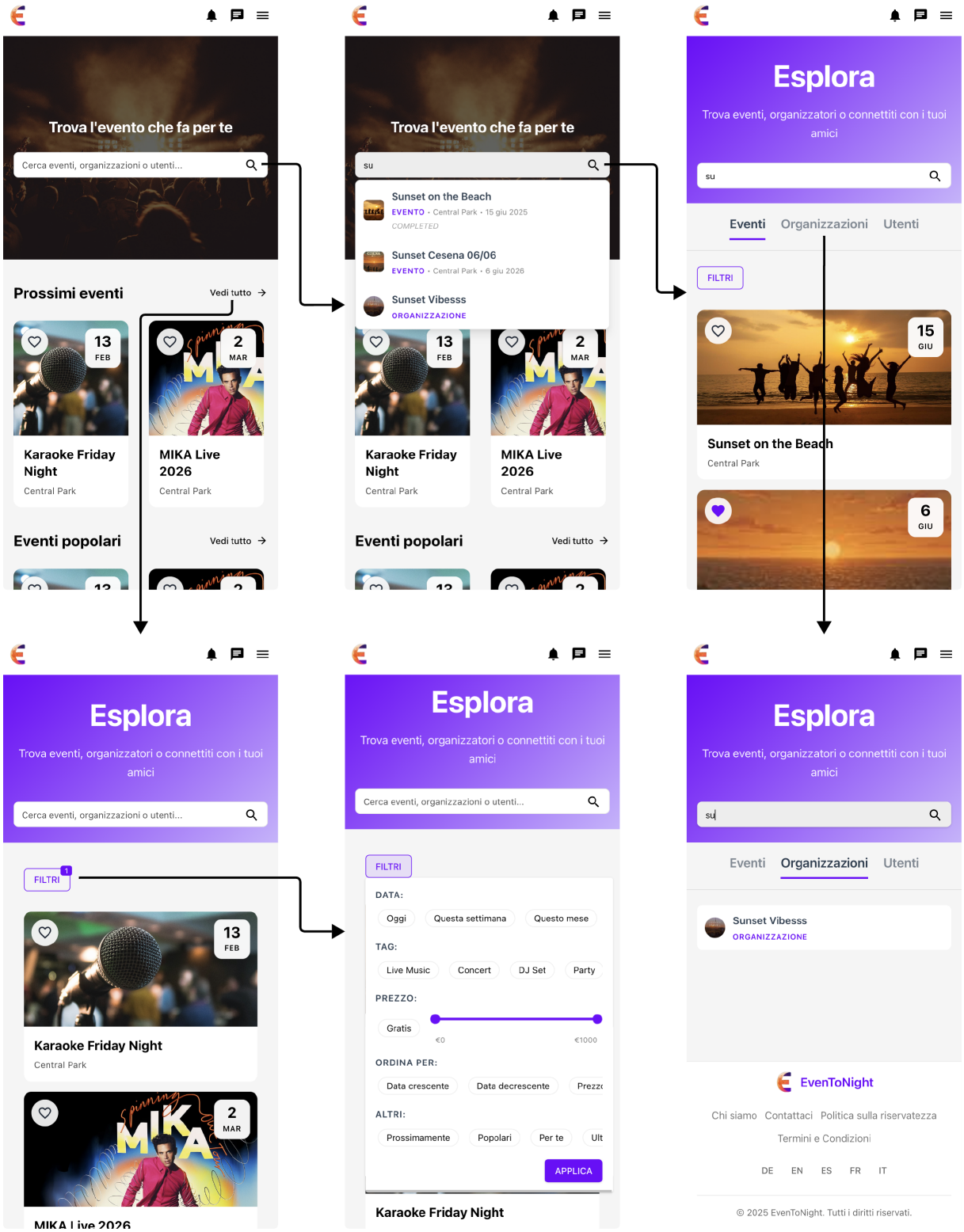

Figure 10: Storyboard: Exploring the platform

8.2 Login and Registration

After opening the application, a user can log in or register to access the additional features offered by the platform.

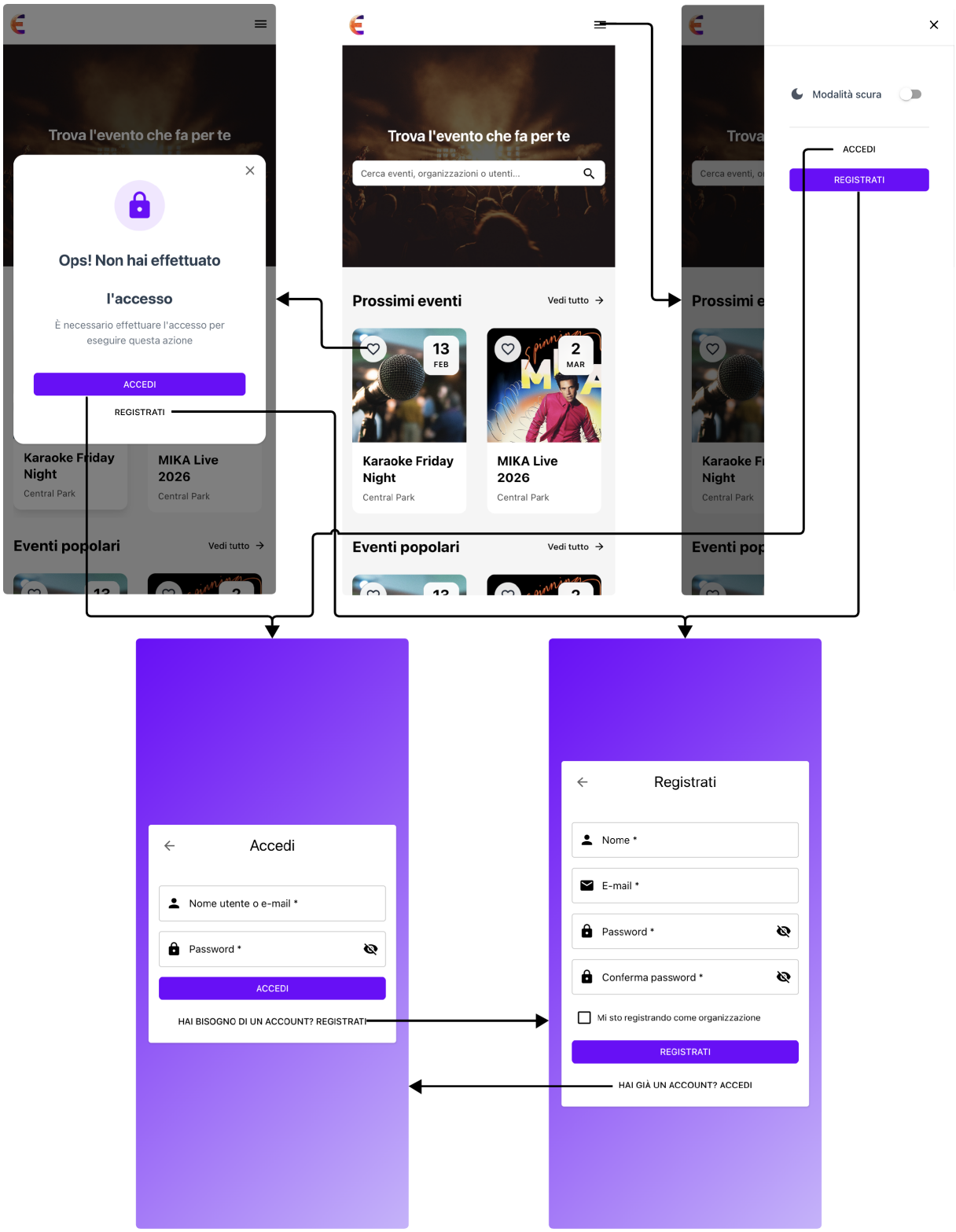

Figure 11: Storyboard: Login and registration

8.3 Create an Event

An organization that has registered on the platform can create events. The creation happens through a form where all the relevant data is entered. It is also possible to temporarily create a draft of the event and continue editing it later.

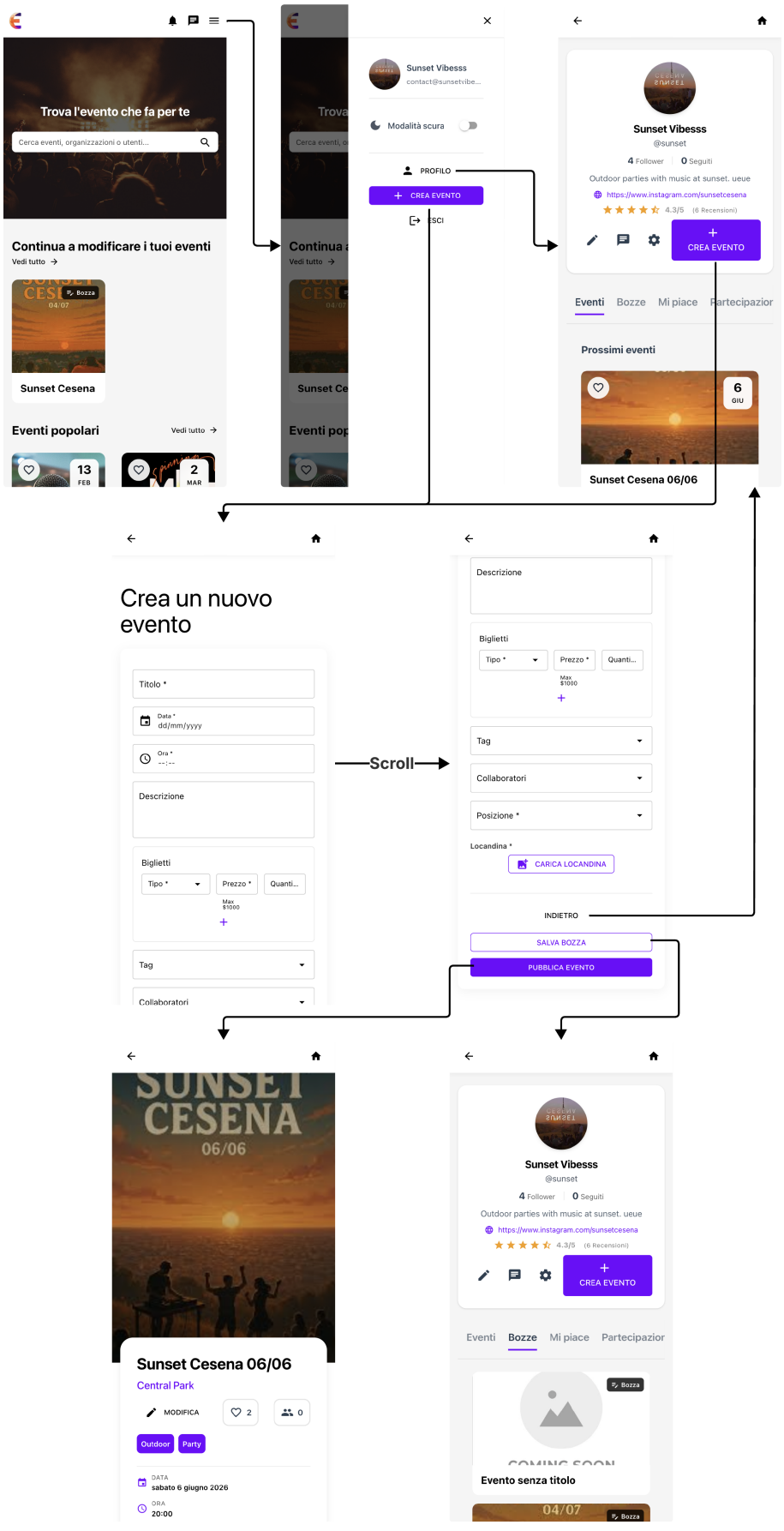

Figure 12: Storyboard: Creating an event

8.4 Attend an Event

A registered user can purchase tickets for different events and view them later. The tickets will contain a QR code that organizations can use to verify them.

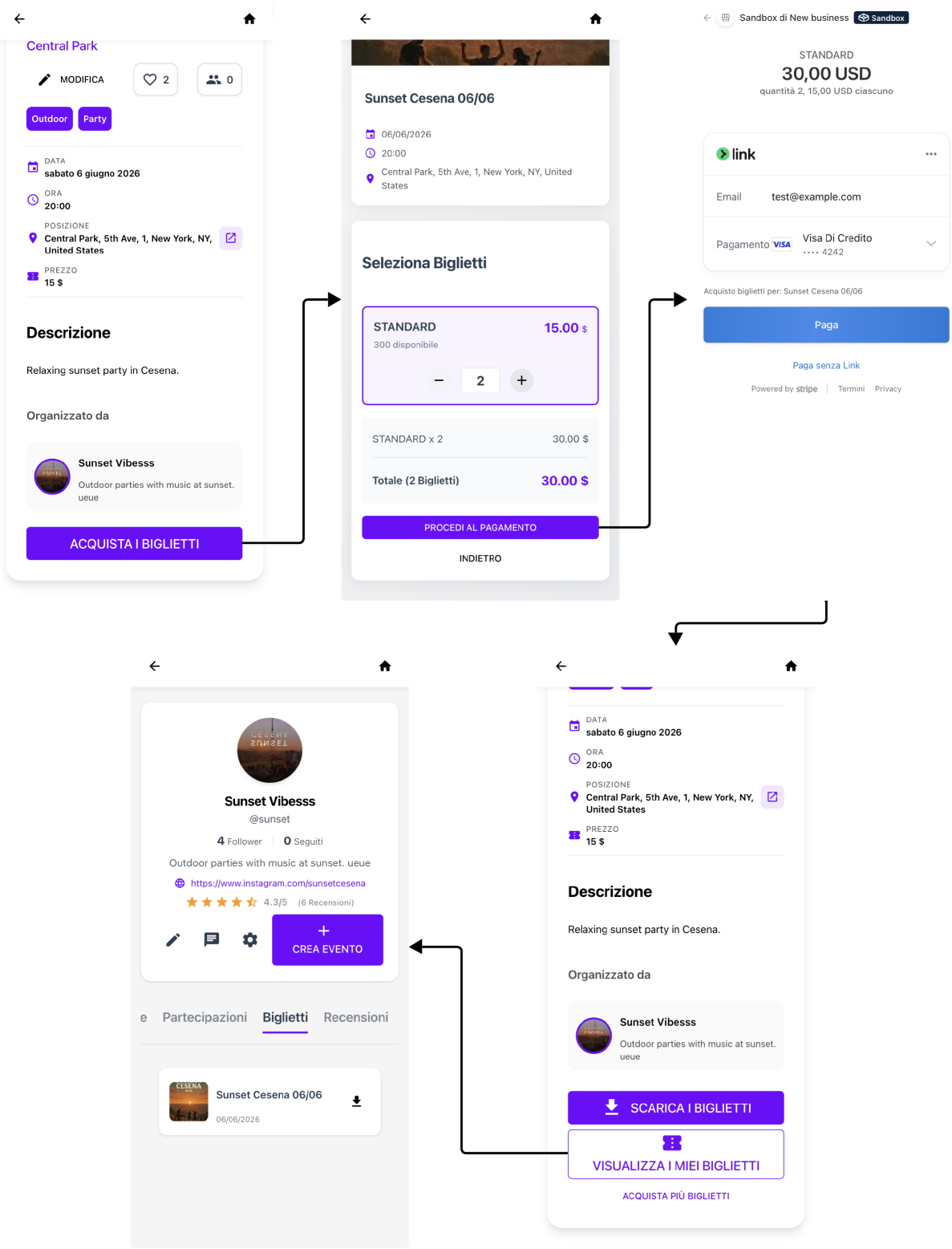

Figure 13: Storyboard: Purchasing a ticket

8.5 Review an Event

After attending an event, a user can decide to leave a review. Once submitted, they cannot leave another review for the same event, but they can modify or delete it.

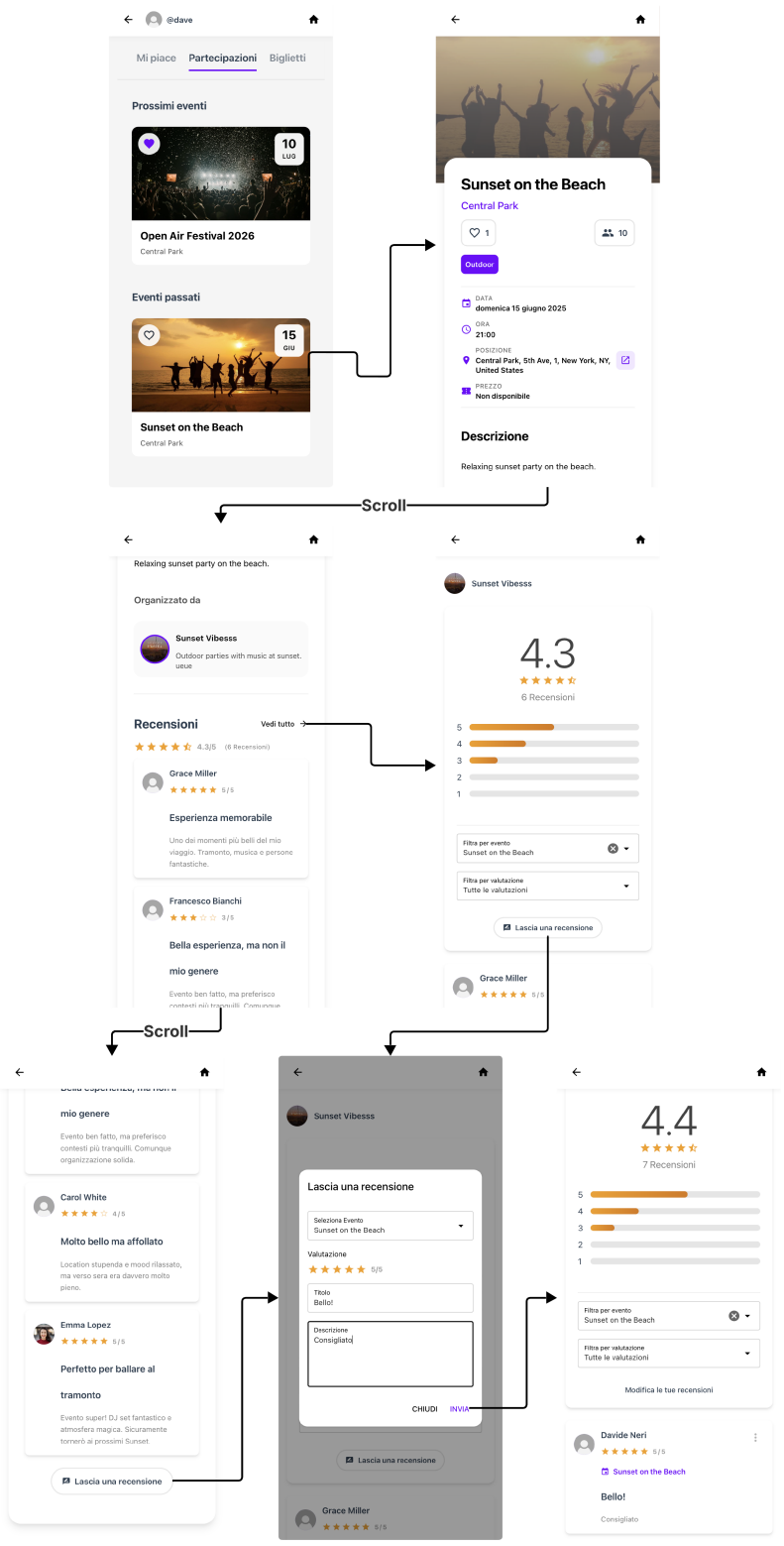

Figure 14: Storyboard: Reviewing an event

8.6 Contact an Organization

A registered user may need to contact an organization to ask for more information or for potential issues.

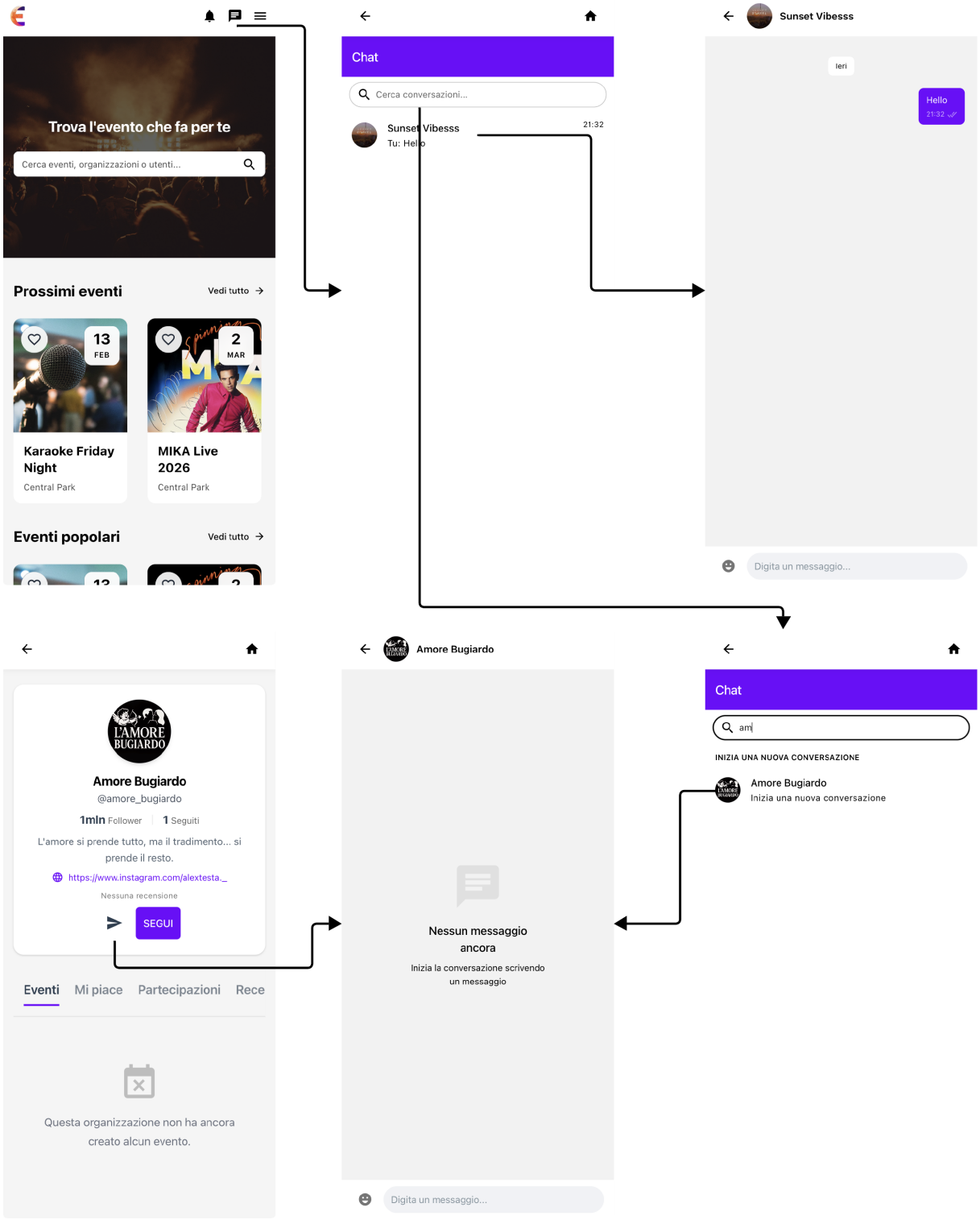

Figure 15: Storyboard: Contacting an organization

9 Conclusions

We are satisfied with the platform developed, which originated from our business idea. The main achievements are:

- **Achievement of project goals.** All project goals were reached, with particular focus on aspects defining a distributed system. The system consists of isolated, loosely coupled components that appear to users as a single coherent platform.
- **Requirements fulfillment.** All functional and non-functional requirements were satisfied, with iterative refinement based on feedback from potential users.
- **Microservices architecture design.** The system was designed as a collection of independent services with clear boundaries. Backend microservices communicate and coordinate via a message broker and expose well-defined REST APIs used by the frontend.
- **Distributed architecture implementation.** Services were containerized using Docker and deployed both with Docker Compose and Docker Swarm, simulating a distributed environment with service isolation and internal networking.
- **Modularity and extensibility.** The architecture provides a clear separation of concerns, ensuring maintainability, easy addition of new features and future scalability. The modular design also facilitated parallel development of different functionalities.

9.1 Future Works

Although the platform was designed according to distributed principles, it was deployed on a single physical machine. As a result, the system does not provide real high availability: if the host machine fails, all services become unavailable. Future improvements could include deployment on multiple physical or virtual machines to achieve infrastructure-level fault tolerance and resilience.
From a functional perspective, potential extensions include:

- Allowing organizations to respond to reviews.
- Additional notifications, such as friendship suggestions and updates on liked events.
- Profile insights and statistics.
- Refund management.
- Multi-currency support to enhance internationalization.

9.2 What We Learned

This project allowed us to apply distributed system concepts that we had previously studied only from a theoretical perspective. In particular:

- We gained practical experience with the trade-offs between consistency and availability in the presence of network partitions, as described by the CAP theorem.

- We experimented with containerization and faced challenges in guaranteeing data consistency across multiple services and system correctness when multiple replicas are deployed.
- We gained a deeper understanding of service decoupling, transparency, reliability and resilience both in single and multiple node deployments.
- We learnt how to make a real online deployment using a proprietary domain and physical server.