EvenToNight

Software Process Engineering - Final Report

Bravetti Federico (federico.bravetti2@studio.unibo.it)
Brini Tommaso (tommaso.brini@studio.unibo.it)

June 12, 2026

Contents

1 Introduction

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.

This report presents the project from the perspective of the **software development process** and the **DevOps** practices adopted throughout its development, including build automation, versioning, quality assurance, deployment and continuous integration / continuous delivery (CI/CD) pipelines.

From a technical perspective, EvenToNight is designed as a **multi-language, containerized and distributed microservices system** and the entire codebase is organized as a **monorepo**, where all services, shared libraries, infrastructure and documentation live in a single Git repository.

1.1 Project Requirements

- **Meaningful Microservices architecture**: design and implement a meaningful division into independent microservices leveraging DDD.
- **Multi-platform**: microservices implemented on at least two different platforms — **Node.js** and the **JVM**.
- **Build automation**: unified build orchestration with **Gradle** across all services and languages.
- **CI/CD pipelines**: automated pipelines covering building, testing, convention enforcement (e.g. Conventional Commits), documentation deployment, Docker image publishing to a registry, and automated deployment to a test environment.

1.2 Process Organization

1.2.1 Version control

- **Git**: the team adopted GitHub Flow — `main` is the only release branch and is protected via branch protection rules; working branches are merged into `main` exclusively via pull request subject to **code review** and automated checks.
- **Commit convention**: all commits follow the **Conventional Commits** specification.
- **Branch convention**: all branches follow the **Conventional Branch** specification.

1.2.2 Build and quality

- **Build system**: **Gradle** is used as the single build orchestrator; each microservice is a Gradle subproject, including Node.js services, which are integrated via the node-gradle plugin.
- **Linting and formatting**: configured per microservice according to its language and ecosystem.
- **Testing**: each microservice has its own test suite, targeting an overall coverage of **70%**.

1.2.3 Release and deployment

- **Semantic versioning**: versions are derived automatically from the commit convention via **semantic-release** .
- **Dockerization**: each service is packaged as a Docker image and published to a container registry (**ghcr.io**).
- **Deployment**: the system must be deployed to a test environment.

1.3 Licensing

EvenToNight is released under the **GNU General Public License v3.0** (GPL-3.0), a **strong copyleft** license: any redistributed modified version must keep the same license, so the source stays open through every chain of redistribution. Unlike permissive licenses such as **MIT** or **Apache 2.0** which allow modifications to be folded into proprietary closed-source products.

Given the academic and open-source nature of this project, this license best reflects the intent to keep the codebase and any derivative works freely available and inspectable.

2 Design

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

The principal objectives of the design phase were:

- service **autonomy** and independent deployability;
- **isolated persistence** per service: no service ever reads or writes another service's database directly;
- **scalability and fault isolation** at the service granularity;
- **local strong consistency** within each service, complemented by **eventual consistency** across services;
- **separation of responsibilities** aligned with the partitioning of the business domain.

To ensure separation of responsibilities, the identification of domain entities and service boundaries has been guided by principles inspired by Domain-Driven Design (DDD).

The platform is articulated into autonomous **bounded contexts**, each one shipped as an independent microservice and communicating with the others either synchronously over HTTP (for request issued by the frontend) or asynchronously through domain events on a shared RabbitMQ broker.

2.1 Domain

The domain of this project is *connecting organizations that promote social events with users interested in discovering and participating in them.*

This domain was modelled through Event Storming, from which the glossary that constitutes the ubiquitous language was then derived.

2.1.1 Event Storming

The DDD model emerges from a sequence of collaborative **Event Storming** sessions, each producing one of the diagrams reproduced below. The sessions were carried out on a shared LucidChart board.

Big Picture Event Storming The first session collected every relevant domain event as a past-tense fact, with no order, command or aggregate attached yet; just the raw vocabulary of the domain, clustered loosely by topic.

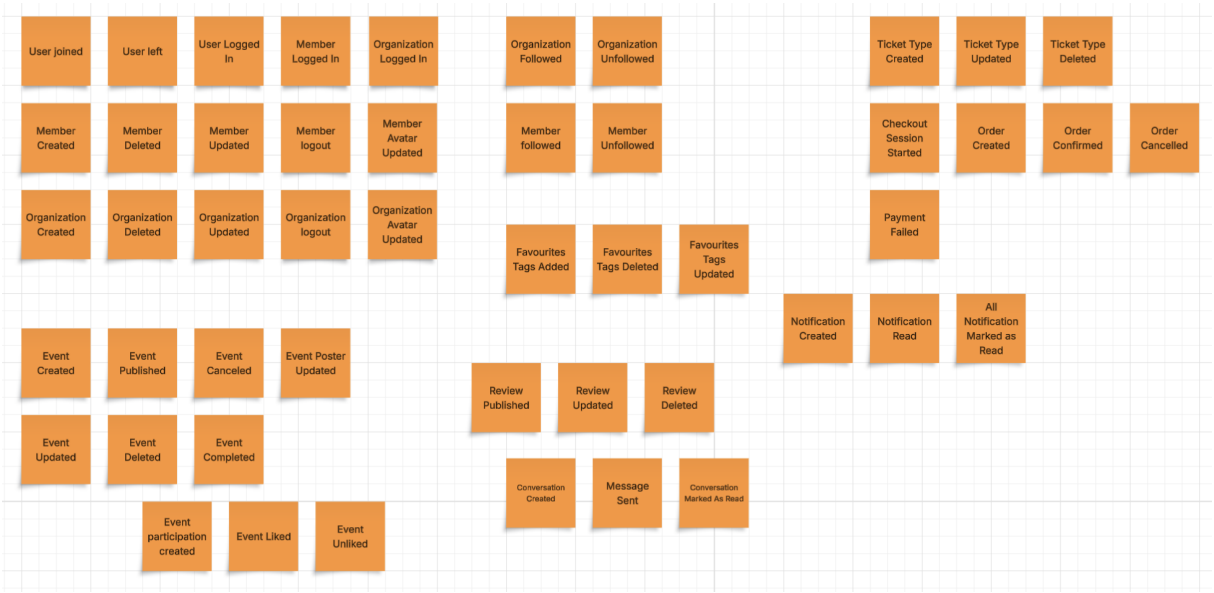


Figure 1: Big Picture Event Storming

Process Modeling Event Storming A second pass ordered the events along temporal arrows, surfaced open questions as **hotspots** (yellow post-it, addressed in the rest of the chapter) and promoted a small subset of events to **pivotal** by drawing a thicker border. Around the pivotal events the team made explicit the **policies** the system runs automatically; the convergence of every cross-context policy on `Notification Created` is the strongest evidence that Notifications deserves to be extracted as a context of its own, a decision finalised at the next level.

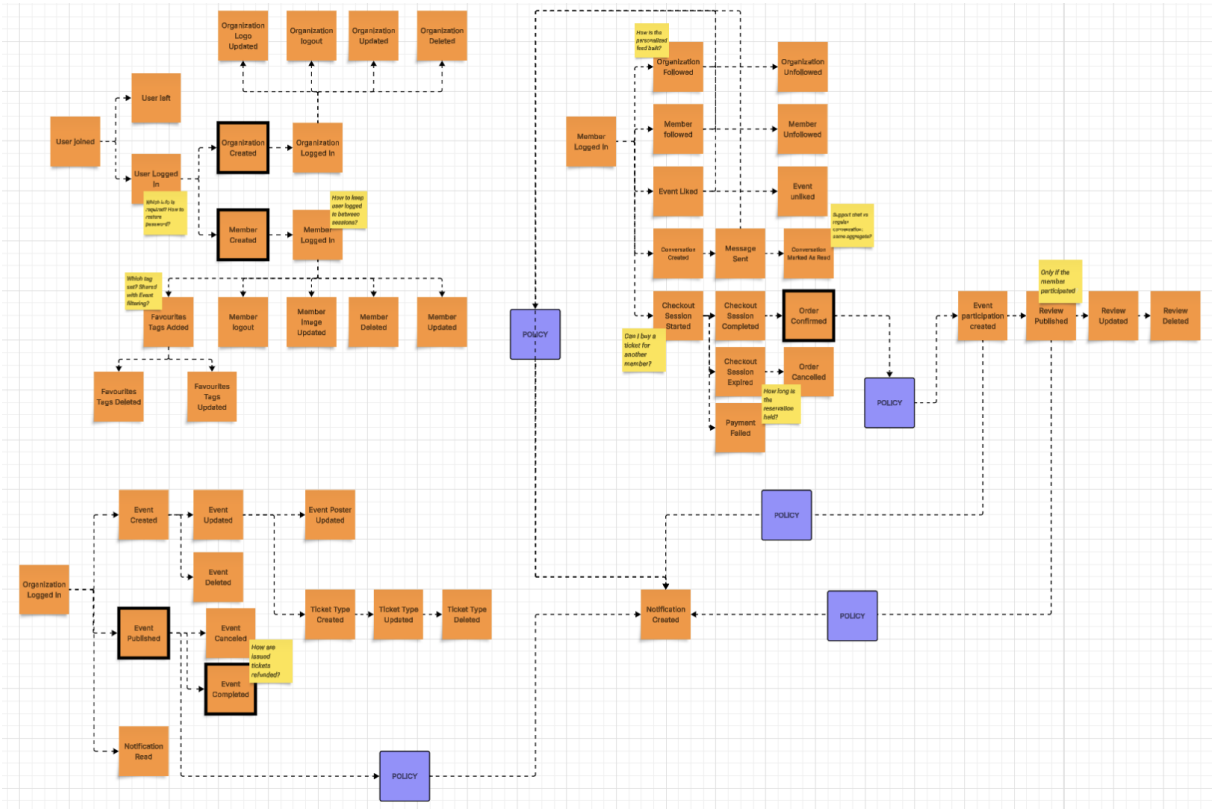


Figure 2: Process Modeling Event Storming

Software Design Event Storming The third pass introduced the only technical element: aggregates. For each command, the team identified the aggregate that validates it and emits the resulting event; clustering aggregates by linguistic cohesion yielded the seven bounded contexts shown in the figure, which became the seven microservices.

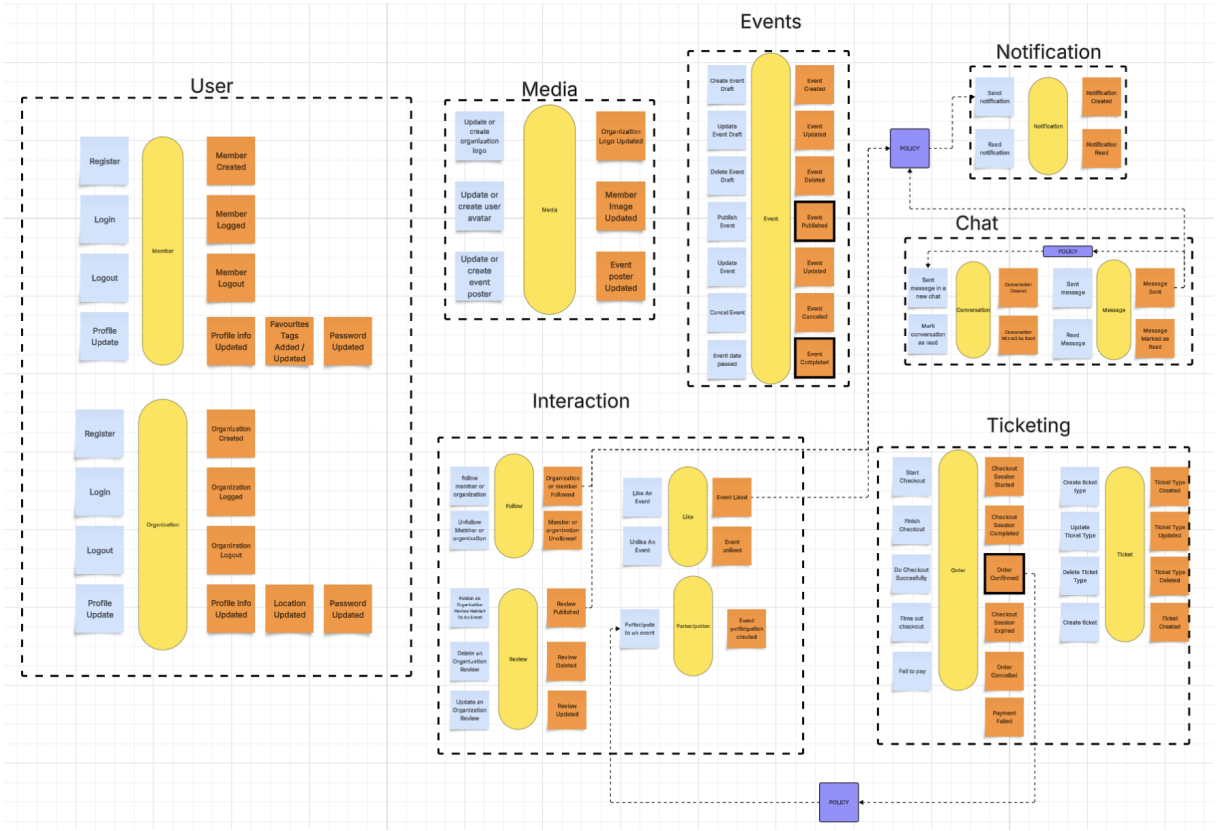


Figure 3: Software Design Event Storming

2.2 Ubiquitous Language

The ubiquitous language is the shared vocabulary that both the team and the code use to describe the domain. The following table collects the terms that recur both in the report and in the source code, with the bounded context that owns each definition. Anywhere two contexts use the same word, they may give it a *different* meaning: that is intentional and is what justifies the translation each context applies when consuming another context's domain events.

Term	**Owner context**	**Meaning**
Member	User	A registered physical person, attendee of events
Organization	User	A registered entity that creates and runs events
RegisteredUser	User	Sealed family that covers both `Member` and `Organization`
Event (entity)	Event	A social event scheduled by an organization; has a lifecycle `DRAFT` → `PUBLISHED` → `(CANCELLED | COMPLETED)`
Collaborator	Event	An organization, other than the creator, that co-hosts an event
EventTicketType	Ticketing	A purchasable tier of tickets for a given event (price, available quantity, sold quantity)

Term	Owner context	Meaning
Ticket	Ticketing	A single seat sold to a single attendee; has its own lifecycle `PENDING_PAYMENT` → `ACTIVE` → (`USED` | `REFUNDED` | `CANCELLED` | `PAYMENT_FAILED`)
Order	Ticketing	The transactional grouping of tickets bought together in one checkout session
CheckoutSession	Ticketing	The Stripe-hosted payment session that confirms or expires an order
Like / Review / Participation	Interaction / Notification	The three ways a member interacts with an event
Follow	Interaction / Notification	The directed relationship `follower` → `followed` between two users
Conversation / Message	Chat	A two-party private chat between any pair of users
Notification	Notifications	A user-facing fact derived from a domain event, possibly delivered in real time
Domain Event	(shared)	A fact, named in past tense, that the publishing context guarantees happened (e.g. `EventPublished`, `OrderConfirmed`)

2.3 Bounded contexts and microservices

The bounded contexts identified in the last event-storming phase are mapped one-to-one to deployable microservices. Each microservice owns its persistence, exposes an HTTP API for synchronous reads issued by the frontend and exchanges domain events with the others through RabbitMQ.

Microservice	Stack	Persistence	Responsibility
`users`	Scala 3 (Cask)	MongoDB + Keycloak	Registration, profile management, authentication tokens
`events`	Scala 3 (Cask)	MongoDB	Lifecycle of `Event`, tags, search, filtering
`ticketing`	NestJS	MongoDB + Stripe	Ticket types, checkout, tickets, orders, PDF / QR generation
`interactions`	NestJS	MongoDB	Likes, reviews, participations, follows, projections of events / users used for cross-cutting validation
`chat`	NestJS + Socket.IO	MongoDB	Real-time private conversations
`notifications`	Node.js (Express) + Socket.IO	MongoDB	Persistent notification feed and real-time push
`media`	NestJS	S3-compatible bucket, MinIo	Generic upload / download of binary assets

2.3.1 Domain Model

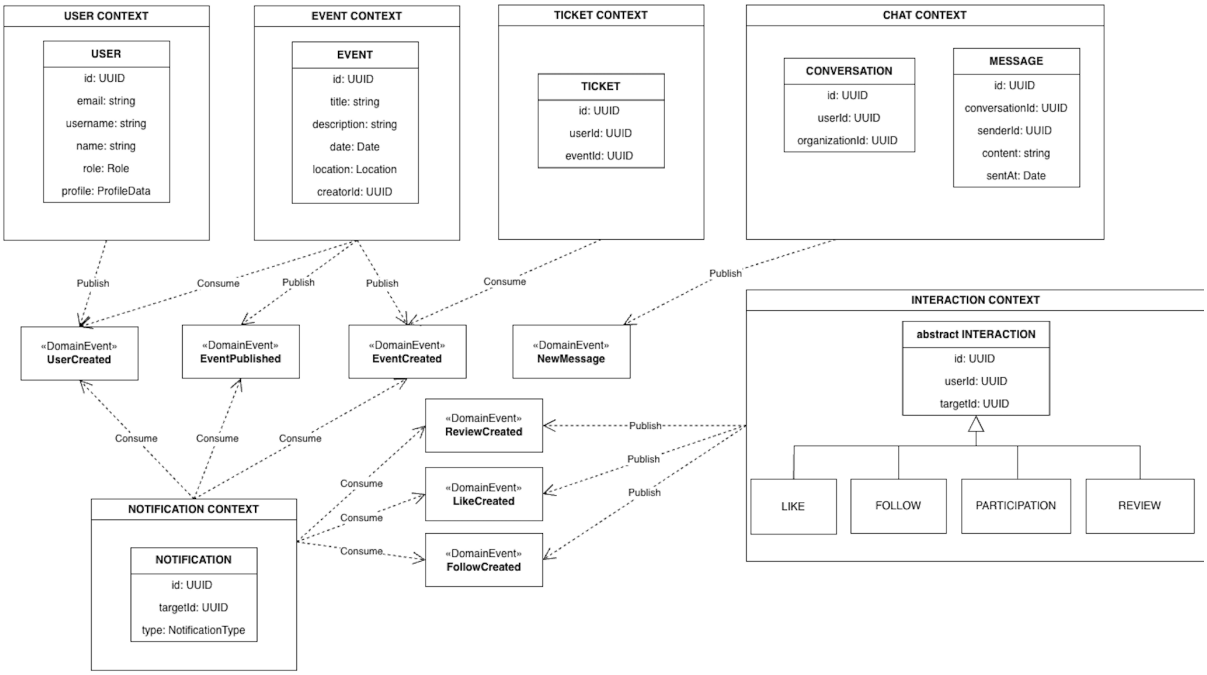


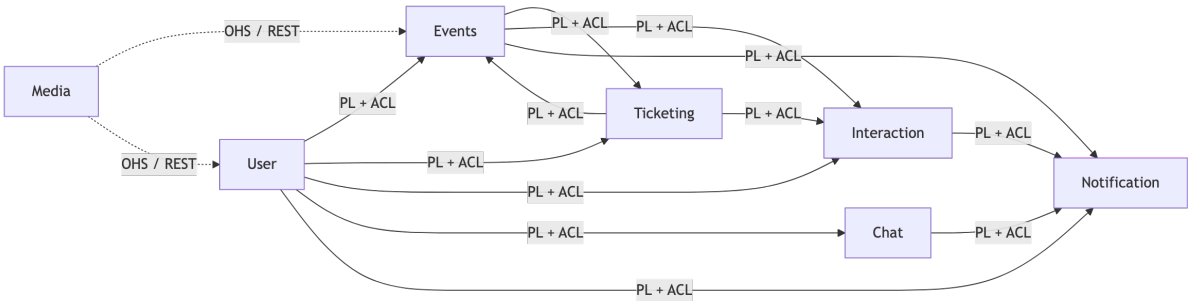
Figure 4: Domain Model overview

2.3.2 Context map

Each arrow goes from the **upstream** context to the **downstream** consumer and is labelled with the integration pattern:

- **PL (Published Language)** — the publisher commits to a stable event envelope and routing-key contract;
- **ACL (Anti-Corruption Layer)** — the consumer translates the upstream payload into its own internal types;
- **OHS (Open Host Service)** — a synchronous REST contract, used by the generic Media context.

Every asynchronous relationship uses both PL and ACL: the publisher owns the contract, the consumer owns the translation.



Three structural observations follow directly from this map:

1. **User is the system's pure upstream** — it publishes user lifecycle events consumed by every other context, but it does not consume events from any other context.

2. **Notification is the system's pure downstream** — it consumes from four different publishers (Events, Interaction, Chat, and User) and emits no business event of its own. This convergence is what justified extracting Notification as a context of its own already at the third Event-Storming level.

3. **Media is the only synchronous integration** — it is invoked over HTTP by the contexts that need to store posters and avatars; the contract is a thin REST API, not a Published Language. Every other inter-context communication is asynchronous over RabbitMQ.

2.4 Integration patterns

This section captures *how* the bounded contexts collaborate at the code level, given that they are independently deployed on different stacks.

2.4.1 No shared domain kernel

A *shared kernel* was deliberately avoided: it would force deployment coupling and language coupling (the kernel must run on every stack). Each bounded context defines its own internal types, with **only the attributes it needs locally**. E.g. The `users` service models a `RegisteredUser` with full profile and account value objects; the `ticketing` service models a `User` as `(UserId, Language)`, enough to localise the ticket PDF.

2.4.2 Shared technical libraries

Only purely technical and domain agnostic code is shared, as two TypeScript packages consumed by the Node services:

- [**`libs/ts-common`**] contains `EventEnvelope`, RabbitMQ publisher, MongoDB transaction manager and `@Transactional()` decorator, Outbox base implementations, pagination and currency helpers.

- **`libs/nestjs-common`** contains NestJS adapters of the above (Mongoose schemas, messaging module, JWT guards).

The Scala services reimplement the same primitives natively, a modest duplication accepted in exchange for stack independence.

2.4.3 Published Language

The contract carried on RabbitMQ has three layers: a hierarchical **routing key** (`<context>.<aggregate>.<verb-past>`, e.g. `event.published`, `payments.order.confirmed`), a common **envelope** defined in `libs/ts-common` and a **payload schema** owned by each publishing context.

```
interface EventEnvelope<T> {
  eventType: string;    // the routing key
  occurredAt: Date;
  payload: T;
}
```

2.4.4 Anti-Corruption Layer

Every consumer implements an ACL that dispatches on the routing key, validates the payload against a local DTO, and maps it into the service's own internal types.

Beyond translation, each ACL also **persists a local projection** of the upstream facts it needs. This is the mechanism that allows services to enforce domain rules that depend on data owned by another context, without issuing synchronous cross-service calls at request time.

2.5 Behaviour

Two behavioural patterns describe how the services process work.

2.5.1 Request-driven operations

User actions follow a request-driven workflow:

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.
4. Optionally after the transaction completes, a domain event is asynchronously published to RabbitMQ.

The full HTTP API contract for each service is documented in the OpenAPI specification.

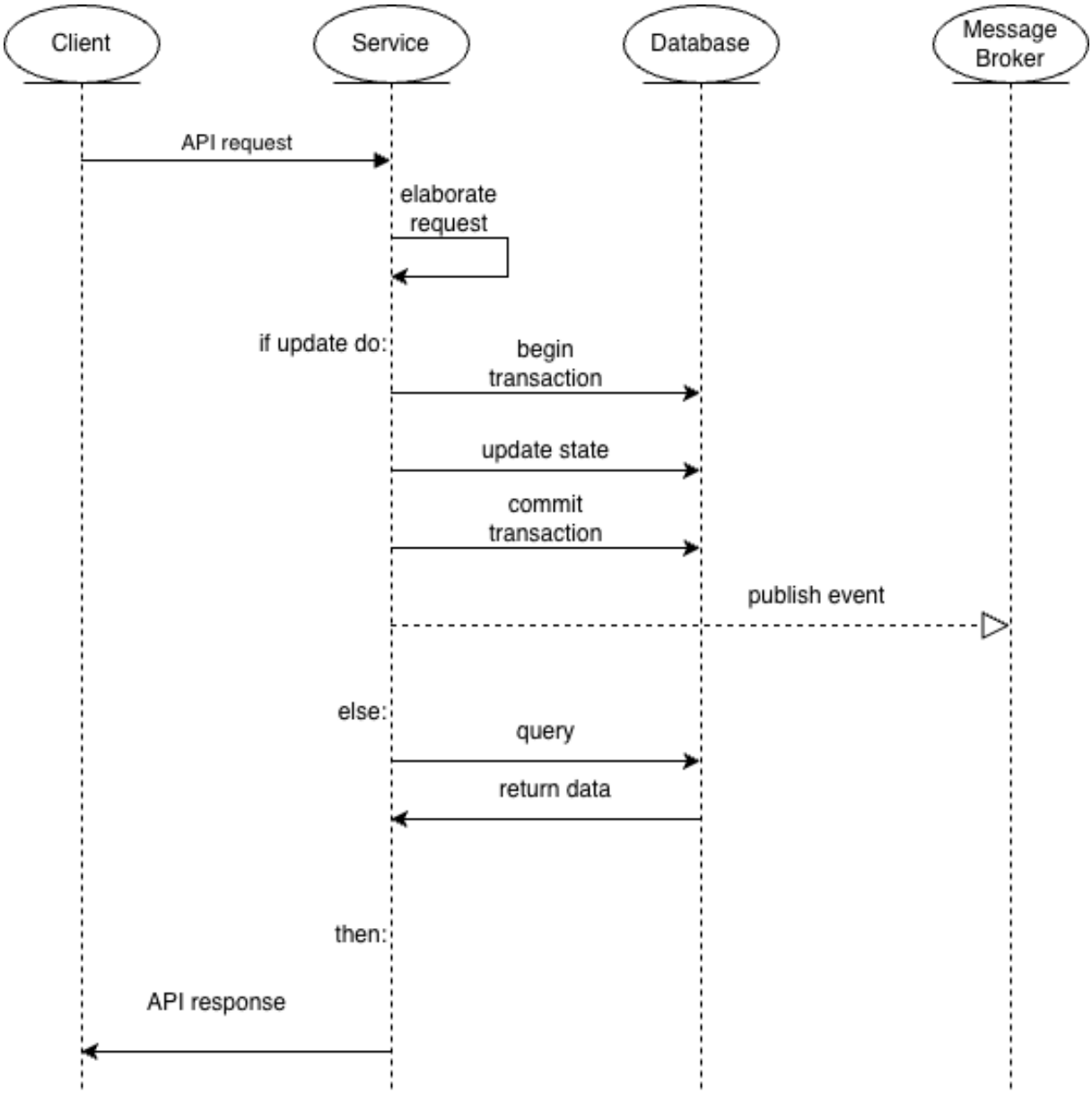


Figure 5: behavior-request-driven

2.5.2 Event-driven operations

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.

The full set of domain events exchanged between services is documented in the AsyncAPI specification.

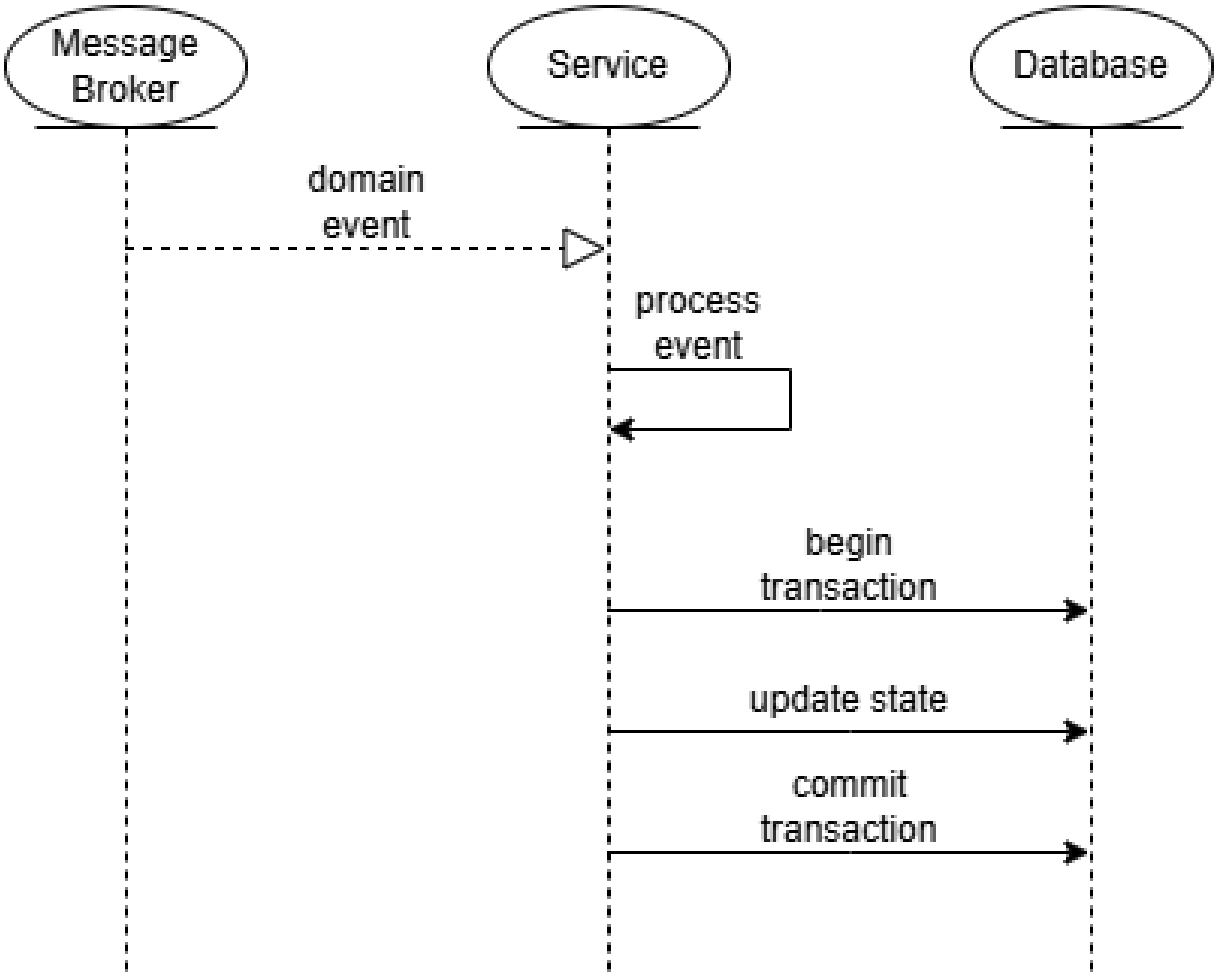


Figure 6: behavior-event-driven

2.6 Internal service architecture

2.6.1 Clean Architecture

The DDD-styled services (`users`, `events`, `ticketing`, `notifications`) are structured following clean architecture structure:

- **`domain/`** — aggregates, value objects, domain events, repository interfaces, domain services.
- **`application/`** — use cases, services and DTOs. Orchestrates the domain without containing business rules of its own.
- **`infrastructure/`** — concrete adapters of the domain ports: MongoDB repositories, RabbitMQ publishers and consumers, Keycloak / Stripe.
- **`presentation/`** (`controller/` in Scala) — HTTP routes, REST controllers, AMQP consumer dispatchers, WebSocket gateways.

The remaining services (`chat`, `interactions`) adopt the default **NestJS module-per-feature** organisation, with each feature module encapsulating its controllers, services and Mongoose schemas.

2.6.2 CQRS (light)

CQRS (Command Query Responsibility Segregation) is an architectural pattern that separates the *write* path (commands that mutate state) from the *read* path (queries that return data), allowing each to evolve, scale, and be optimised independently.

In this project CQRS has been adopted in *Events* and *Notifications* only at the structural level: the write and read paths are syntactically separated into distinct handler classes, but they ultimately share the same MongoDB collections. No separate read store or projection pipeline has been implemented. The separation is therefore primarily a code-organisation choice.

2.7 Distributed consistency

Each service guarantees strong consistency within its own boundaries by executing state-changing operations inside local MongoDB transactions. Coordination across services is achieved through eventual consistency: a service emits domain events after completing its local transaction, and downstream services update their own state asynchronously upon receiving them. No distributed transaction spanning multiple services is ever required.

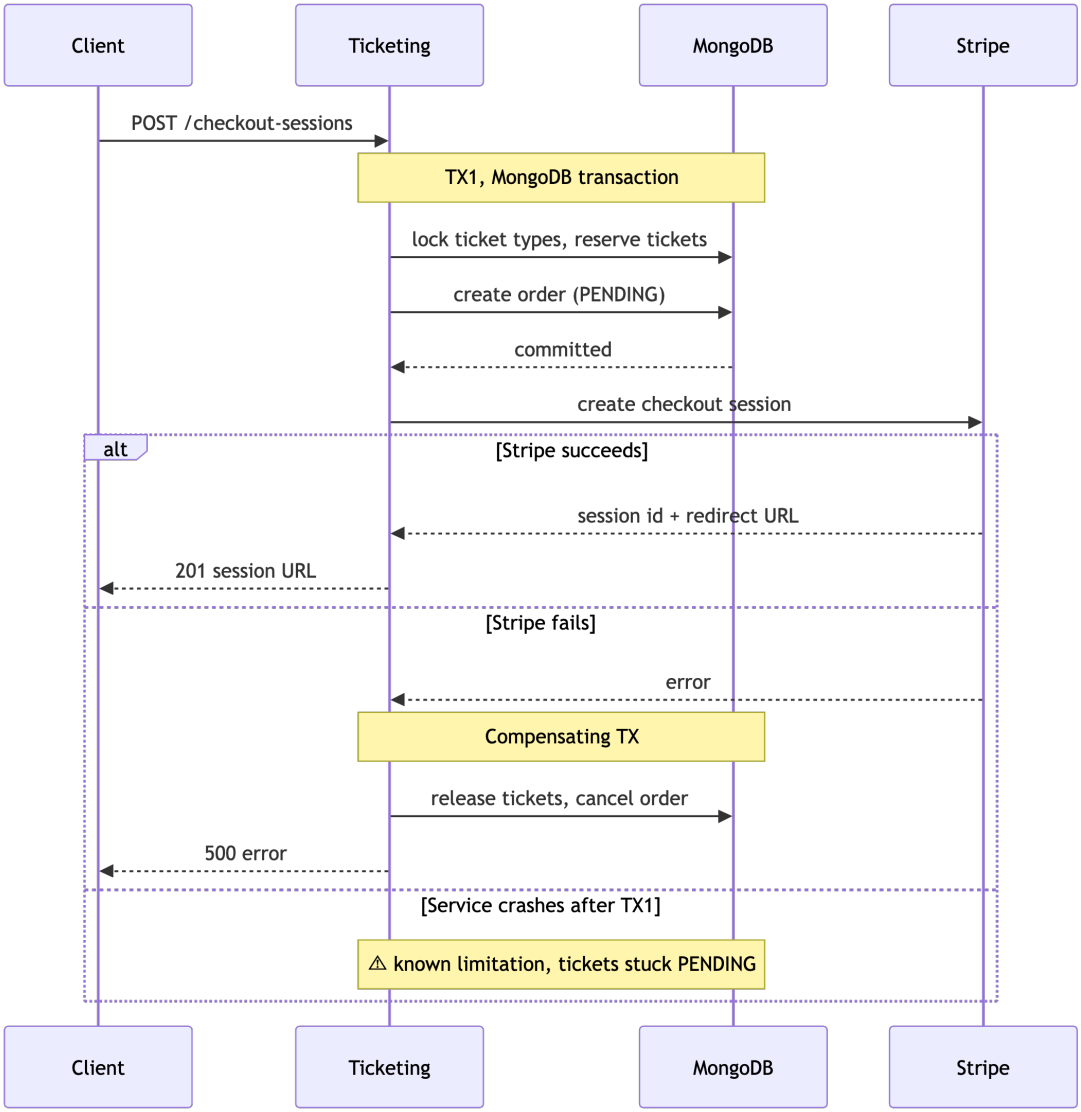
2.7.1 Outbox pattern

Publishing a domain event directly after a database write is not atomic: if the service crashes between the two operations, the state change is committed but the event is never delivered, leaving downstream services in a stale state with no indication that anything went wrong.

To eliminate this risk, every service that emits domain events uses the **Outbox pattern**: the event is written to an outbox collection *within the same local transaction* as the state update. A background process then reads the outbox and forwards the events to RabbitMQ, guaranteeing at-least-once delivery.

2.7.2 Choreographed saga: the ticket purchase

The ticket purchase flow cannot be completed in a single local transaction because it spans both MongoDB and an external system (Stripe). It is therefore implemented as a two-phase **choreographed saga**:


Client
Ticketing
MongoDB
Stripe
POST /checkout-sessions
TX1, MongoDB transaction
lock ticket types, reserve tickets
create order (PENDING)
committed
create checkout session
alt
[Stripe succeeds]
session id + redirect URL
201 session URL
[Stripe fails]
error
Compensating TX
release tickets, cancel order
500 error
[Service crashes after TX1]
⚠ known limitation, tickets stuck PENDING
Client
Ticketing
MongoDB
Stripe

3 DevOps

3.1 Build Automation

3.1.1 Gradle as a multi-language orchestrator

Gradle is the project's primary build system, set up as a single multi-project build. The root holds two files: `build.gradle.kts`, defining project-wide tasks (environment setup, Docker environment orchestration and git pre-commit helpers), and `settings.gradle`, which registers every subproject and configures the git hooks.

Each service is a Gradle subproject, but the build spans different languages. JVM/Scala services (`events`, `users`) build natively through Gradle, while JS/TS services (`chat`, `interactions`, `media`, `notifications`, `ticketing` and the `frontend`) are wrapped via the node-gradle plugin, which delegates the actual work to npm. A single Gradle invocation thus drives heterogeneous toolchains behind one uniform task interface.

The monorepo layout was preferred because it enables:

- **unified build and CI**: one Gradle invocation builds, checks and tests the whole system, and a single CI pipeline validates every service;
- **frictionless code sharing**: the libraries under `libs/` are consumed directly by the Node services, with no external registry publishing;
- **single versioning** for the whole product;
- **simplicity for a small team**: one repository, one build and one set of conventions keep the maintenance and cognitive overhead low.

3.1.2 Convention plugins and custom tasks

To avoid duplicating build logic, `buildSrc/` defines two **convention plugins**, one for the Scala services and one for the JS/TS ones. Each encapsulates the language-specific compilation, style and coverage setup and exposes it under a common set of task names.

A custom `ExecTask` further simplifies running arbitrary shell commands, with sequential chaining, `onSuccess`/`onFailure` callbacks and built-in cross-platform support. Shared constants are also defined alongside it under `buildSrc/`.

3.1.3 Git hooks

Hooks are installed through the gradle-pre-commit-git-hooks plugin configured in `settings.gradle`. Two **pre-commit** hooks are registered:

- `formatAndLintPreCommit` runs the format-and-lint task (which every subproject configures via its convention plugin), then re-stages the files that were staged at commit time, so any style fix is included in the commit;
- `updateAndCheckEnvSetup` updates `.env` from `.env.template` (adding any missing variable) and verifies that both files share the same keys and that `.env` values are populated, so a variable added only to the local `.env` cannot be pushed without also being declared in the shared `.env.template`.

A **commit-msg** hook additionally validates the commit message against the Conventional Commits convention.

3.2 Dockerization

3.2.1 Dockerfile design

Every service ships its own `Dockerfile`, designed around two goals: **layer caching** and **minimal image size**. Instructions are ordered properly for maximising cache reuse, and each image uses a **two-stage build**: a builder stage compiles the artifact while the final stage carries only the runtime plus the built output.

3.2.2 Docker Compose layer system

Every service and infrastructure component defines its Compose files, which are not standalone: utility scripts (`findComposeFiles.sh`, `composeAll.sh`) discover them and **merge them in layers**, each adding a concern on top of the previous one:

- **base** (`docker-compose-base.yml`) — the canonical service definition: image, networks, environment and healthchecks;
- **main** (`docker-compose.yaml`) — production-oriented overrides: restart policy, Traefik routing labels and `depends_on` conditions;
- **dev** (`docker-compose-dev.yaml`) — development extras: local image `build`, exposed host ports and dev tooling (e.g. mongo-express);
- **swarm** (`docker-compose-swarm.yaml`) — Swarm-specific settings: `deploy` (replicas, placement constraints, restart policy), overlay networks and configs.

A standard deploy merges `base` → `main`; a development environment adds `dev` on top; a Swarm deploy (beta) merges `base` → `swarm` instead of the main file.

3.3 DVCS

3.3.1 Git Flow

The repository follows a Git-Flow-inspired strategy: a single long-lived branch, `main`, updated exclusively through a **pull request** from feature branches. The merge strategy depends on the PR: small changes whose individual history carries little value are **rebased** to keep the history linear, while larger feature branches are integrated with **merge commits**, preserving the branch topology.

3.3.2 Branch protection

`main` is protected by a set of rules:

- changes must arrive through a pull request **reviewed and approved by at least one other team member**;
- the required CI status checks must pass before merging;
- direct pushes to `main` are blocked; the only whitelisted identity is a dedicated **GitHub App**, whose token lets the release workflow push the generated CHANGELOG.

Pull requests to `main` also enable automatic **Copilot review**, which gives an immediate first-pass on every PR, flagging obvious bugs and smells before a human reviewer steps in.

3.3.3 Environment and secrets

The `.env.template` file is the authoritative schema for the environment. Configuration variables are given a default value, while API keys and passwords are left empty. The `checkEnvSetup` task leverages this convention to enforce two properties:

1. it **fails on empty values**, forcing real secrets to be provided locally and removing the risk of shipping placeholder credentials;
2. it **fails on key mismatches** between `.env` and `.env.template`, keeping the schema in sync.

To still allow building and testing in CI the pipeline injects default passwords before the check runs. Variables and secrets actually needed by the workflows are stored as **GitHub Actions Variables/Secrets**.

3.4 CI/CD Pipelines

3.4.1 Quality gates on pull requests

Three workflows act as quality gates on every pull request to `main`:

- `build-and-test.yml` runs `./gradlew clean build` across all services (compile, check and test) and uploads coverage reports to **Codecov**; on pull requests it additionally detects which `Dockerfiles` changed and builds those images, validating that they build cleanly before the code lands.
- `check-style.yml` runs `./gradlew checkStyle` to verify linting and formatting.
- `check-commit-convention.yml` inspects every commit in the PR and validates it against the Conventional Commits specification.

Style and commit-convention checks are also enforced as git hooks, but they are rechecked in CI because hooks can be bypassed locally.

3.4.2 Translation check

The `auto-i18n.yml` workflow runs the project's own `auto-i18n` action on pushes to the frontend feature branch (`feature/frontend-service`) and on every pull request. On push it generates the missing translations and commits them; on pull requests it runs in **check-only** mode. To avoid re-triggering the pipeline on the auto-generated commit, the commit message carries `[skip ci]`, which is omitted when a PR is already open from that branch, so that all required checks still run.

3.4.3 GitHub Pages

The `deploy-pages.yml` workflow triggers on pushes to `main` and to any `docs/*` branch. It builds the three VitePress report sites (ASW, DS, SPE), copies the static assets and the OpenAPI/AsyncAPI specifications into the output tree, and publishes everything to **GitHub Pages**.

3.4.4 Release and Deploy

Every push to `main` triggers the `release.yml` pipeline, which first decides whether a release is needed:

- **semantic-release** inspects the commits and publishes a new versioned release following Conventional Commits + SemVer;
- if no release is published but `services/` or `infrastructure/` changed, the pipeline falls back to an incremental **dev pre-release** (e.g. `v1.4.0-dev.2`), so that every meaningful change still produces a deployable tag.

When a tag is produced, the pipeline finds all modified directories containing a `Dockerfile`, then builds and pushes the corresponding **multi-arch images** (`linux/amd64` + `linux/arm64`, via Buildx and QEMU) to `ghcr.io`. The service name is derived from the Dockerfile path (e.g. `services/chat/Dockerfile` → `chat`). Each image is pushed under two tags: the **release version** (e.g. `chat:2.2.2`) and `:latest`.

The deployed production infrastructure is the following:

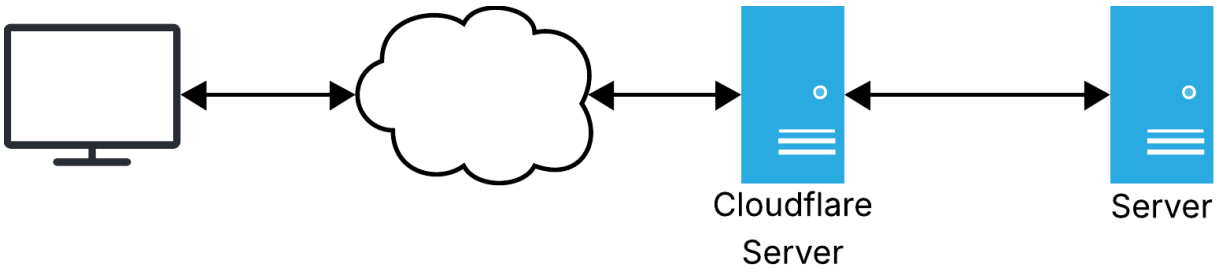


Figure 7: Deployed infrastructure

The deploy job then opens an SSH session to the production server through a **Cloudflare Tunnel**: the connection is authenticated via Cloudflare Access and bridged through Cloudflare's network, so no SSH port is exposed publicly. On the remote machine the repository is pulled, the new images are pulled from the registry and the stack is redeployed via `composeApplication.sh`.

A **Docker Swarm** deployment path is also available (still in beta): `scripts/swarmDeploy.sh` manages the full Swarm lifecycle, including pushing the images to Docker Hub so they are reachable from every swarm node.

4 Translation Action and CLI Tool

Internationalising the frontend was a project requirement: the application is offered in five languages: English, Italian, Spanish, French and German. Keeping five translation files aligned by hand is tedious and error-prone, so **English was chosen as the source of truth** and a dedicated tool, `auto-i18n`, was developed to keep the other locale files in sync and to verify their completeness.

The tool lives in an **independent repository** and is distributed in two complementary forms built around a single shared core:

- a **GitHub Action**, invoked from the CI pipeline (see DevOps) to translate on push and to gate pull requests;
- an **npm CLI** (`@eventonight/auto-i18n`), for running the same translation locally during development.

5 Conclusion

We are satisfied with the engineering process we designed and applied to the EvenToNight platform. The project gave us the chance to put into practice process-engineering and Domain-Driven Design concepts that we had previously studied only in theory.

What we learned

- **Build automation across stacks.** A single Gradle multi-project orchestrates build, test and quality checks for both Scala and Node.js services with one invocation, keeping the pipeline indifferent to the internal structure of each service.
- **Continuous Integration / Continuous Delivery.** Dedicated pipelines take care of integration and deployment, automating the path from a code change to a running deployed release, while making us proficient with the more advanced features of Git and GitHub.
- **Documentation and tooling.** Reports and API specs are built and deployed automatically alongside the code.
- **Domain Driven Design.** We applied Event Storming to analyze the domain and align on a shared ubiquitous language, which then guided the modeling of our system as a microservice architecture.