Politehnica University of Bucharest

Software Engineering

Bask IT Up!

Software Requirements Specification

Team:

Cristiana Florentina PRECUP, 1241EB
Vlad Andrei DUMITRU, 1241EA
Florin Mihai PRODAN, 1241EA

Coordinator:

Prof. Dr. Ing. Nicolae GOGA

Date created:

Monday, October 20, 2025

Delivery Report

(will be delivered along with the project)

Name	Group	Project implementation [%, reason]	Signature
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____

Delivery date:

_____

Table of Contents

Requirements Analysis

According to the IEEE STD-830-1993, *IEEE Recommended Practice for Software Requirements Specification*.

1. Introduction

1.1. Purpose

This Software Requirements Specification (SRS) document defines in detail both the functional and non-functional requirements for **BaskIT**, an e-commerce web application designed to offer curated, themed gift baskets tailored to the Romanian market. The document establishes a clear, shared understanding of the project's scope, objectives, and expected quality standards among all stakeholders—namely the course staff and product owner—as well as the development team encompassing frontend, backend, and DevOps members, and the testers responsible for validation and verification.

By formalizing the product vision and requirements, this SRS acts as the primary reference for system design, implementation, and maintenance, ensuring that all contributors work toward the same technical and business goals. It also supports long-term traceability between business needs, technical design decisions, and implemented features.

1.2. History

BaskIT is a completely new software product, built from the ground up with no preexisting system or legacy components. This document represents **Version 1.0** of the SRS, prepared on **October 18, 2025**, at the inception of the project to define the requirements of the Minimum Viable Product (MVP).

Future revisions of this SRS will capture evolving requirements and improvements as additional integrations (such as expanded shipping providers or enhanced payment features) are finalized and as new features are proposed, including the optional **"build-your-own basket"** functionality. The history of changes, versions, and authors will be documented in the *Document Evolution* appendix to maintain transparency and continuity throughout the development lifecycle.

1.3. Scope

BaskIT is an e-commerce platform that enables customers to browse, select, and purchase **themed gift baskets**, each composed of carefully curated items around specific occasions or interests (such as holidays, celebrations, or personal hobbies). Customers can explore baskets through an intuitive storefront, add items to their cart, and securely complete their purchases using **Stripe** as the payment processor.

The platform handles a complete order lifecycle—**Created → Paid → Fulfilled → Canceled/Refunded**—ensuring data consistency, transactional safety, and traceable audit entries for all key actions. Inventory is managed **per basket**, allowing administrators to prevent overselling and maintain stock accuracy at all times.

The application distinguishes between two administrative roles:

- A **Content Manager**, responsible for maintaining the catalog by creating, editing, and publishing baskets, images, prices, and descriptions; and
- An **Admin**, a super-user with complete control over both operations and content management, including order tracking, shipping label generation, system configuration (VAT, shipping methods), and user account management for content managers.

The MVP version of **BaskIT** targets a **mobile-first experience** optimized for modern browsers and focuses on the Romanian market, using **RON** as the currency and supporting configurable **VAT** rates. Security and reliability are ensured through **JWT-based authentication**, **webhook signature verification**, structured logging, and **idempotent webhook processing**. These technical measures, combined with a clear separation of roles and permissions, provide a robust foundation for future scalability and maintainability.

1.4. Definitions, Acronyms and Abbreviations

Term / Acronym	Definition
Basket	Curated product bundle sold as a single item.
Order	Customer purchase containing one or more baskets and totals.
Shipment	Delivery associated with an order (carrier, service, tracking number, label URL).
Admin	Authenticated back-office user with full privileges.
Customer	Guest or registered user placing orders.
Content Manager	Back-office user with CRUD access over products but no access to orders or system settings.
Inventory (per basket)	Available stock units for each basket SKU.
RON	Romanian leu (currency).
VAT	Value Added Tax applied to qualifying sales.
AWB	Shipment label identifier used by couriers.
Stripe Payment Intent	Stripe object representing the lifecycle of a payment attempt.
Webhook	Server-to-server callback notifying BaskIT about external events (e.g., payment succeeded).

1.5. References

[1] K. E. Wiegers and J. Beatty, “Software Requirements,” 3rd ed., Microsoft Press, 2013.
[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software,” Addison-Wesley, 1994.
[3] E. Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software,” Addison-Wesley, 2003.
[4] M. Fowler, “Patterns of Enterprise Application Architecture,” Addison-Wesley, 2002.
[5] S. Newman, “Building Microservices,” 2nd ed., O’Reilly Media, 2021.
[6] M. Kleppmann, “Designing Data-Intensive Applications,” O’Reilly Media, 2017.
[7] R. C. Martin, “Clean Architecture: A Craftsman’s Guide to Software Structure and Design,” Prentice Hall, 2017.
[8] M. Richards and N. Ford, “Fundamentals of Software Architecture,” O’Reilly Media, 2020.

[9] M. Nygard, “Release It!: Design and Deploy Production-Ready Software,” 2nd ed., Pragmatic Bookshelf, 2018.
[10] J. Humble and D. Farley, “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation,” Addison-Wesley, 2010.
[11] N. Madden, “API Security in Action,” Manning Publications, 2020.

1.6. Structure

This SRS is organized to guide the reader from context to detail and to ensure that every requirement is traceable to design artifacts and tests.

- **Section 1 — Introduction** defines the purpose, history, scope, terminology, references, and the overall document structure. It establishes a common vocabulary and the boundary of the MVP.
- **Section 2 — General Description** summarizes the product at a high level (what the system does and for whom). It describes the product functions, user types, constraints, and key assumptions/dependencies that influence the solution.
- **Section 3 — System Requirements** captures the verifiable requirements that drive design and testing:
    - **3.1 External Interface Requirements** describe UI surfaces, programmatic APIs, software/hardware interfaces, and communication protocols.
    - **3.2 Functional Requirements** list feature-level behaviors using “shall/should” statements grouped by domain (catalog, checkout, payments, shipping, admin, etc.).
    - **3.3 Performance Requirements** define expected responsiveness and throughput targets, plus qualitative performance goals.
    - **3.4 Design Constraints** specify mandated technologies, regulatory constraints (e.g., GDPR/PCI boundaries), and architectural decisions that limit design options.
    - **3.5 Software System Attributes** collect non-functional qualities such as security, reliability, maintainability, usability/accessibility, observability, and portability/compatibility.
    - **3.6 Other System Requirements** includes data model highlights, business rules, and acceptance samples that do not fit elsewhere.
- **Appendices (A1–A9)** provide supporting material and detailed artifacts referenced by Sections 2–3:
    - **A1. Interview with the customer** — discovery notes capturing stakeholder goals and constraints.
    - **A2. System diagram(s)** — context, deployment, and component diagrams showing boundaries, runtime topology, and major building blocks.
    - **A3. Use Case Diagrams** — actor-centric overviews for **Customer**, **Content Manager**, and **Admin**; each use case maps to functional requirements.
    - **A4. Class Diagrams** — domain model used across documents; class names match those in requirements for consistency.
    - **A5. Sequence Diagrams** — key flows (e.g., checkout, label generation) that illustrate interactions over time.

- **A6. State Diagrams** — object lifecycles (e.g., Order, Shipment) and valid transitions.
- **A7. Document Evolution** — version history with dates, authors, and a summary of changes.
- **A8. Report regarding team meetings** — agendas, decisions, and action items to demonstrate process and coordination.
- **A9. Conclusions regarding the activity** — reflections on progress, risks, and next steps.

Reading order. Instructors can skim Sections **1–2** to grasp scope and users, review **3.2/3.5** for functional and non-functional depth, and then consult **Appendices A2–A6** for diagrammatic evidence. Testers should use **3.2/3.3/3.6** as the basis for acceptance criteria and test plans; designers and developers should rely on **3.1–3.5** together with **A2–A6** for architecture and implementation guidance.

Traceability. Functional requirements in **3.2** reference corresponding diagrams in **A3–A6**; the data model in **3.6** matches classes in **A4**; performance and reliability targets in **3.3** inform test scenarios in the appendices. This structure ensures each requirement can be traced to design artifacts and, ultimately, to verification activities.

2. General description

2.1. Product Description

BaskIT is a web-based e-commerce system that sells curated, themed gift baskets in Romania. The product exposes two user-facing interfaces: (1) a customer storefront (mobile-first, browser-based) for browsing, cart, checkout, and order tracking; and (2) an admin dashboard for managing baskets, categories, prices, images, inventory (per basket), orders, and shipments.
Externally, BaskIT integrates with **Stripe** for payments (Checkout/Payment Intents + webhooks) and a **pluggable courier interface** for generating shipping labels and storing tracking numbers. The system persists data in PostgreSQL and sends transactional emails via an SMTP/email service provider. All client–server communication uses HTTPS/JSON.

System Goal and Vision: BaskIT aims to simplify the gifting process by offering a curated digital marketplace where users can select, purchase, and ship themed gift baskets in minutes. Its design emphasizes convenience, security, and maintainability—serving as both a course-level project in software engineering best practices and a model for scalable e-commerce architectures.

2.2. Product Functions

At a high level, BaskIT provides:

- **Catalog & discovery:** browse by category, search/filter baskets, view basket details (images, description, price in RON, stock).
- **Cart & checkout:** add/update/remove items, compute totals (items, VAT, shipping), guest checkout with optional account creation, secure card payment via Stripe, order confirmation.
- **Orders & shipping:** order status tracking (Created → Paid → Fulfilled → Canceled/Refunded), shipment creation after payment, tracking number and label URL displayed to the customer.
- **Admin operations:** CRUD for baskets/categories/images/prices, stock adjustments (per basket) with audit trail, view/filter orders, change order status, generate shipping labels, review audit logs of sensitive actions.

Together, these four domains establish a complete transactional workflow—from product discovery to fulfillment and audit. Each functional area in this section directly maps to the system requirements in Section 3.2 and to the diagrams in the appendices, ensuring consistency between business goals, technical design, and implementation.

2.3. User description

- **Customers:** visitors purchasing gifts. They access the storefront on modern mobile and desktop browsers. No special technical knowledge is required. Accessibility considerations (adequate contrast, keyboard navigation, alt text) are provided where feasible. Guest checkout is supported; registered customers may view order history.
- **Content Manager**: back-office user responsible for website content. Has CRUD access to products (baskets), including titles, descriptions, images, categories, prices, and stock values. Can preview changes before publishing. All actions are recorded in the audit log.
- **Admin**: super-user for shop operations. Has full privileges, including everything a Content Manager can do, plus order and shipment management, inventory adjustments, viewing customer/order data, configuration (VAT/shipping/provider keys), audit log review, and CRUD over Content Manager accounts (create/update/disable/reset). Typically works on desktop with basic familiarity with back-office tools.

Relationship between Roles: Admins oversee Content Managers and may adjust their access levels or reset credentials, but Content Managers cannot modify Admin privileges. Both roles share the same authentication system, differentiated by assigned permissions. Customers interact exclusively with the public storefront and have no direct contact with administrative modules.

2.4. Constraints

- **Regulatory & compliance:** GDPR for personal data; card data handled by Stripe (no raw card data stored by BaskIT). VAT is configurable; prices displayed in **RON**.
- **Technical environment:** web application (Chrome/Edge/Firefox/Safari, last two major versions); server-side Java (Spring Boot) with PostgreSQL; outbound internet required for Stripe, email, and courier APIs.
- **Interfaces with other applications:** Stripe payments and webhooks; a chosen courier/aggregator API for labels/tracking; SMTP/email provider.
- **Operational constraints:** not safety-critical, but **financial and inventory operations must be correct** and idempotent (e.g., payment → order status → stock updates).
- **Capacity & parallelism (initial target):** expected up to ~100 concurrent customer sessions and 1–3 concurrent admin users; stock updates use transactional control to avoid overselling; webhook handlers are idempotent to tolerate retries.

Portability and Safety Considerations: The system is designed to be portable across containerized Linux and Windows environments, with full compatibility on mobile and desktop browsers. While not safety-critical, all financial operations are safeguarded by transactional integrity and secure communication protocols to prevent data loss or unauthorized access.

2.5. Assumptions and Dependencies

- **Runtime & platform:** Linux container host; Java 21 (Spring Boot), Node.js for frontend build, PostgreSQL 14+; HTTPS with a valid TLS certificate.
- **Third-party services:** active Stripe account (test → live), selected courier/aggregator credentials, SMTP/email service.
- **Network & infrastructure:** stable outbound connectivity to third-party APIs; DNS/domain configured for storefront and webhook endpoints.
- **Project scope assumptions:** Romanian market at launch; **two administrative roles (Admin and Content Manager)**; ‘build-your-own basket’ and advanced recommendations are post-MVP extensions.

Dependency Impact: Any prolonged downtime or change in third-party services (such as Stripe API updates or courier outages) may temporarily limit order processing. The system’s modular adapter design mitigates this risk by allowing quick substitution of providers without architectural changes.

3. System Requirements

3.1 External Interface Requirements

The **BaskIT** platform interacts with its users and external systems primarily through web interfaces and standardized RESTful APIs. These interfaces ensure that both customers and administrators can interact with the system efficiently, securely, and consistently across multiple platforms and devices.

3.1.1 User Interfaces

BaskIT exposes two primary web-based user interfaces, each designed with a specific audience and purpose in mind:

1. **Storefront (Customer-Facing Interface)**
   The storefront is a responsive, **mobile-first** application accessible through standard web browsers such as Chrome, Edge, Firefox, and Safari. It includes all the functionality necessary for end-to-end shopping:
   - **Home Page:** Highlights featured baskets and promotions.
   - **Catalog View:** Displays product categories in a grid layout with filter and search capabilities.
   - **Basket Detail Page:** Shows basket contents, images, descriptions, and stock information.
   - **Cart and Checkout:** Allows customers to manage their selections, compute totals (including VAT and estimated shipping), and complete secure payments.
   - **Order Confirmation & Tracking Pages:** Display order summaries, statuses, and shipment tracking numbers after purchase.
   - **Authentication Pages:** Support for optional customer accounts to view order history.
   - **Error Pages:** Custom error handling for missing resources (404) and general system errors (500).
   - **Legal & Compliance Pages:** Cookie consent, privacy policy, and terms of service.
2. The design emphasizes clarity, visual consistency, and accessibility, adhering to basic **WCAG** standards for color contrast, keyboard navigation, and alternative text.
3. **Administrative Dashboard**
   The administrative interface is optimized for desktop use and provides **role-based access** to internal users.
   - **Content Managers** access catalog management features, including CRUD operations for baskets, categories, and product images.
   - **Admins** can additionally view and update orders, manage inventory, generate shipping labels, configure VAT and courier integrations, and monitor the audit log.
     Each administrative action is securely authenticated and logged for accountability.

3.1.2 Application Programming Interfaces (REST/JSON)

All system communication between frontend and backend components occurs via **RESTful APIs** over secure HTTPS connections. Responses and requests are encoded in **JSON**, ensuring compatibility and simplicity for integrations.

Administrative endpoints require **JWT-based authentication**, while public endpoints such as catalog browsing remain open for anonymous users.

Representative API Endpoints:

- **Public APIs:**
  - GET /api/baskets?category=&q=&page= — Retrieve a paginated list of baskets with optional filtering.
  - GET /api/baskets/{slug} — Fetch detailed information for a single basket.
- **Cart & Checkout APIs:**
  - POST /api/cart/estimate — Compute cart totals and shipping estimates.
  - POST /api/checkout/session — Create an order and initialize a Stripe payment session.
- **Webhook Endpoint:**
  - POST /api/webhooks/stripe — Receive asynchronous payment notifications from Stripe.
- **Admin Authentication APIs:**
  - POST /api/auth/login — Obtain JWT tokens after credential verification.
  - POST /api/auth/refresh — Renew session tokens.
- **Admin Catalog & Order APIs:**
  - POST/PUT/DELETE /api/admin/baskets{,/id} — Manage basket data.
  - GET /api/admin/orders — Retrieve all orders for processing.
  - PATCH /api/admin/orders/{id}/status — Update an order's status.
  - POST /api/admin/shipments/{orderId} — Generate shipment labels.

Each endpoint follows REST conventions and supports standard HTTP response codes (200, 201, 400, 401, 404, 500) to convey operational results.

3.1.3 Hardware Interfaces

No specialized hardware is required beyond standard server or cloud infrastructure capable of hosting web services, relational databases, and containerized applications. The system may run in virtualized or physical environments, using modern x86-64 architecture.

3.1.4 Software Interfaces

BaskIT interacts with a limited set of third-party and internal software systems:

- **Stripe:** Handles all card payments via Checkout and Payment Intents APIs with webhook callbacks for transaction updates.
- **Courier Provider:** A pluggable REST-based adapter for AWB generation, shipping label creation, and tracking number retrieval.
- **SMTP/Email Service:** Used to send transactional emails, such as order confirmations and password resets.
- **PostgreSQL Database:** Primary data store for catalog, orders, users, and audit logs.
- **Redis (optional):** In-memory cache for session management and short-lived data.

Each external system is encapsulated behind a well-defined adapter, preserving flexibility and facilitating future replacements without significant code changes.

3.1.5 Communications Interfaces

All communications between system components and external providers use **HTTPS (TLS 1.2+)** to ensure encryption and data integrity.
Webhook endpoints verify provider signatures (e.g., Stripe's Stripe-Signature header) before processing to prevent tampering or replay attacks.
Outbound connections are established for courier APIs and email services. Inbound traffic is restricted to HTTPS ports (443) with firewall and rate-limiting protections in place.

3.2 Functional Requirements (Traceable to A3 Use-Case Diagrams)

> All functional requirements below are represented one-to-one in **Appendix A3** (Customer, Content Manager, Admin), with the same IDs and wording.

Customer

- **FR-UCC1 (List / filter / search):** The system shall list active baskets with title, representative image, price in RON, and stock badge, and shall allow filtering by category and text search in titles/descriptions.
- **FR-UCC2 (View basket details):** The system shall display basket details (images, description, contents, stock quantity) and shall disable **Add to Cart** when the basket is out of stock.
- **FR-UCK1 (Maintain cart & totals):** The system shall maintain a per-session cart and compute totals (items, VAT, shipping estimate) in real time; stock shall be validated on add and on cart updates.
- **FR-UCK2 (Checkout & pay):** On checkout, the system shall create an Order with status **Created**, initiate secure payment and store the Payment Intent ID, then show a success page on payment confirmation or an appropriate message on cancellation/failure.

Content Manager

- **FR-UCA3C (Create catalog entities):** Content Managers shall be able to create baskets, categories, images, and associated pricing/stock fields.
- **FR-UCA3U (Update catalog entities):** Content Managers shall be able to edit baskets, categories, images, pricing, and stock values.
- **FR-UCA3D (Delete catalog entities):** Content Managers shall be able to delete baskets, categories, and images (subject to referential integrity rules).
- **FR-UCA3P (Preview before publish):** Content Managers shall be able to preview catalog changes in a non-public view prior to publishing to the storefront.

Admin

- **FR-UAA1 (Authenticate – JWT):** Administrative users (Admin and Content Manager) shall authenticate with username/password; the system shall issue JWT access/refresh tokens.
- **FR-UAA2 (Manage orders & shipments):** Admins shall search/filter orders, update order status, generate shipping labels via the configured courier adapter, store tracking numbers/label URLs, and retry failed label creation without duplicates; tracking links shall be visible on the customer order page.
- **FR-UAA3 (Adjust inventory – audited):** Admins shall adjust stock levels manually; every change shall be recorded in the audit log with actor, timestamp, and delta.
- **FR-UAA4 (Manage CM accounts):** Admins shall manage Content Manager accounts (create, update, disable, reset passwords).

3.3 Performance Requirements

The BaskIT platform prioritizes **responsiveness, reliability, and scalability**. Users must experience a fluid, uninterrupted interaction across all major workflows, including browsing, checkout, and administration.

The **frontend and backend** must operate harmoniously to minimize perceived latency. Catalog and basket detail responses should typically render within **300 milliseconds** for 95% of user requests under up to 100 concurrent sessions. The objective is to ensure that browsing feels instantaneous, preventing users from abandoning the site due to slowness.

During the **checkout phase**, all internal processing — from order creation to payment intent generation — should complete in under one second, excluding external gateway latency. The checkout sequence must remain reliable even under degraded network conditions or partial API delays.

Webhook processing and asynchronous events shall complete promptly and idempotently, typically within **three seconds**, with duplicate events safely ignored. The system logs processing results and retries transient failures automatically to maintain data consistency between BaskIT and Stripe.

When generating **shipping labels**, variability in courier response times is expected. The system handles such cases gracefully, displaying progress indicators to admins and surfacing meaningful, human-readable error messages if a label cannot be generated immediately. Retrying must never result in duplicate shipments.

Beyond raw speed, overall **availability and reliability** define BaskIT's performance. The platform targets **99% uptime** (excluding maintenance windows), with automated monitoring and alerting for key services. Incident response follows these targets:

- **Critical issues** (checkout or payment failure): immediate acknowledgement and fix.
- **High-priority issues** (admin operations blocked): resolved within 24 hours.
- **Minor defects or content errors:** resolved within 24–48 hours.

BaskIT's architecture supports **portability** and **scalability**, allowing deployment on containerized Linux or Windows environments. Horizontal scaling through additional instances ensures sustainable performance as traffic grows.

Performance validation includes automated **load testing, stress testing, and recovery scenarios**, emphasizing not just metrics but the perceived smoothness of user experience.

3.4 Design Constraints

1. **Technology Stack (DC-1):** Frontend built in React + TypeScript; Backend implemented in Java 21 using Spring Boot; Database layer in PostgreSQL; optional Redis caching.
2. **Architecture (DC-2):** Monolithic codebase employing a hexagonal (ports-and-adapters) structure; security handled through JWT tokens.
3. **Regulatory Compliance (DC-3):** Full GDPR adherence; no card data stored on BaskIT servers (PCI scope delegated to Stripe).
4. **Internationalization (DC-4):** Prices fixed in RON; VAT stored in configuration for easy adjustment.
5. **Time and Locale (DC-5):** All timestamps stored in UTC; displayed in Europe/Bucharest time zone.
6. **Data Integrity (DC-6):** Financial records immutable; instead of deletions, use status transitions.
7. **Deployment (DC-7):** System packaged as containerized services; outbound internet access required for payments, email, and shipping APIs.

3.5 Software System Attributes

Security

All communication must occur over **HTTPS (TLS 1.2+)**. Secrets such as API keys and signing tokens must never appear in source code and are injected through environment variables or configuration vaults. Passwords are hashed using modern algorithms (BCrypt or Argon2). Webhooks are verified for authenticity, and personally identifiable information (PII) is excluded from logs.

Reliability & Availability

All critical workflows — payments, order updates, and stock management — must be **atomic** and **transactional**. The system should tolerate webhook redeliveries and courier API delays gracefully. Automatic retries and monitoring ensure that transient issues recover without human intervention.

Maintainability & Extensibility

Modules follow a **ports-and-adapters** approach to facilitate substitution of third-party providers. Database migrations are applied through versioned scripts to maintain consistency. Unit and integration tests validate each core process, using test doubles for external dependencies.

Usability & Accessibility

The user interfaces must remain intuitive, visually consistent, and accessible on a wide range of screen sizes. Minimum accessibility compliance targets include proper labeling, focus management, color contrast, and descriptive alt text. All error messages must be user-friendly, avoiding technical jargon.

Observability

System logs must include structured metadata such as correlation IDs to trace transactions end-to-end. The audit log must record who performed which action, when, and what data was changed, preserving both accountability and compliance.

Portability & Compatibility

BaskIT must run seamlessly on modern **Linux** distributions and maintain compatibility with **Windows** for development environments. The frontend operates across major desktop and mobile browsers, ensuring a consistent experience regardless of platform.

Safety & Operational Policies

While BaskIT does not control physical or safety-critical processes, it manages financial operations that must be safeguarded against data loss. Monitoring tools will detect anomalies, and incidents will follow an operational response window of **24–48 hours**, depending on severity.

3.6 Other System Requirements

OSR-1: Data Model Highlights

Key entities include:

- **Basket:** id, slug (unique), title, description, price_amount, currency, stock_qty, status, images[], timestamps.
- **Order:** id, customer_email, shipping_address, total_amount, currency, vat_amount, payment_status, order_status, stripe_payment_intent_id (unique), placed_at.
- **OrderItem:** id, order_id (FK), basket_id (FK), title_snapshot, unit_amount, qty.
- **Shipment:** id, order_id (FK), carrier, service, tracking_no (unique), label_url, status.

- **AuditLog:** id, actor (user/IP), action, entity_type, entity_id, before (JSON), after (JSON), occurred_at.

Each record includes metadata for traceability. Relationships are enforced through foreign keys, and referential integrity is guaranteed by PostgreSQL.

OSR-2: Business Rules

- Slugs for baskets and categories must be unique.
- Shipment tracking numbers and payment intent IDs must be unique to prevent duplication.
- Monetary rounding follows consistent decimal precision (two decimal places).
- Orders cannot transition to *Fulfilled* or *Refunded* without passing through *Paid*.

OSR-3: Acceptance Samples

- **AC-01:** When an order in *Created* receives a successful payment webhook, it must transition to *Paid*, decrement stock, send an email confirmation, enable shipment creation, and record an audit entry.
- **AC-02:** When a basket's stock quantity equals zero, the "Add to Cart" option is disabled, and checkout cannot proceed with that item.
- **AC-03:** When the same payment success event arrives twice, the system processes it once only; duplicate deliveries are logged as idempotent replays with no side effects.

Appendices

A1. Interview with the customer

Date: October 18, 2025
Location: Online (Microsoft Teams)
Duration: 1 hour 15 minutes

Participants:

- **Cristiana-Florentina Precup** — *Product Owner & Project Coordinator*
- **Florin Mihai Prodan** — *Backend Developer (Architecture and Database Design)*
- **Vlad Andrei Dumitru** — *Frontend Developer (User Interface and Experience)*

Objective:
The goal of the meeting was to define the business and technical vision of **BaskIT**, elicit both

functional and non-functional requirements for the MVP, clarify user roles and responsibilities, and align the development team on priorities, technology choices, and delivery expectations.
The discussion also served to capture the Product Owner's expectations regarding usability, maintainability, and scalability, forming the foundation for the Software Requirements Specification (SRS).

Meeting Summary

Cristiana-Florentina Precup, acting as **Product Owner**, began the session by presenting the project vision:

> "BaskIT should make gift-giving effortless — a curated platform where anyone can find and order themed gift baskets in minutes."

She explained that the system should simplify the buying process for customers while maintaining professional management tools for internal users. The application must serve two purposes simultaneously: deliver an elegant, seamless shopping experience for customers, and provide efficient operational tools for the administrative staff.

The group agreed on the following **core objectives** for the MVP:

- Allow customers to browse curated gift baskets, view prices, and check availability.
- Support a full checkout process with secure online payment via **Stripe**.
- Maintain real-time inventory tracking to prevent overselling.
- Enable administrative management of baskets, categories, and orders through a structured dashboard.
- Integrate with Romanian courier APIs (e.g., **FanCourier**, **SameDay**, **eColet**) for label generation and shipment tracking.

Cristiana stressed the need for a clean, modular architecture that can evolve beyond the MVP. She proposed a **hexagonal (ports-and-adapters)** design to isolate business logic from integrations, making it easy to replace or extend components such as couriers or payment providers in the future.

Mihai described the backend's architecture and database design principles. He emphasized **transactional consistency** for order creation and stock updates, relying on **PostgreSQL's ACID guarantees**. He also proposed detailed audit logging of every critical event (e.g., order transitions, stock modifications, courier label creation) to ensure traceability and accountability.

Vlad focused on the **frontend experience**, highlighting the importance of a **mobile-first React application**. The interface must be responsive, intuitive, and visually clean. He recommended following accessibility standards (keyboard navigation, alt text, color contrast) and emphasized simplicity in checkout to minimize cart abandonment.

The participants collaboratively defined **user roles** and their responsibilities:

- **Customer:** browses, selects, and purchases baskets via the storefront.
- **Content Manager:** manages catalog content — creates and updates baskets, prices, images, and stock.
- **Admin:** full operational control — oversees orders, shipments, user accounts, VAT configuration, and audit logs.

Cristiana confirmed that clear role separation would help ensure data safety and operational efficiency. Each administrative action should be authenticated using JWTs and recorded in the audit log.

They then discussed **external integrations**. Stripe was selected as the payment solution because it provides easy integration, RON support, and PCI compliance. The courier interface would be implemented as a **pluggable adapter**, allowing future support for multiple providers without altering business logic.

Regarding **non-functional expectations**, the team established:

- The system must run securely over **HTTPS/TLS 1.2+**.
- **GDPR** compliance is mandatory for all customer data.
- The platform should ensure **99 % uptime**, excluding maintenance.
- Errors and incidents must be addressed within **24–48 hours**, depending on impact.
- The system must remain **portable** across Linux and Windows environments using containerization (Docker).
- All interfaces must be responsive and functional on major browsers and devices.

Cristiana concluded the meeting by summarizing the **success criteria** for the project:

1. A stable and reliable MVP capable of completing full orders end-to-end.
2. Secure integration with third-party services using verifiable webhooks.
3. Clear auditability and transparency of operations.
4. A modular architecture that supports future evolution (e.g., build-your-own baskets, promotional codes, or multi-language features).

Decisions and Outcomes

Topic	Agreement / Decision	Responsible
Payment processing	Integrate **Stripe** using Checkout and Payment Intents APIs; verify webhooks for authenticity.	Mihai
Shipping provider	Implement a **courier adapter**; begin with FanCourier or eColet as first integrations.	Mihai
System roles	Introduce two administrative levels: **Admin** (full control) and **Content Manager** (catalog management).	Cristiana
Frontend design	Mobile-first React interface emphasizing clarity and accessibility.	Vlad
Audit and traceability	All sensitive operations logged with before/after data and timestamps.	Mihai
Data integrity	Use PostgreSQL with ACID transactions to prevent overselling and maintain consistency.	Mihai
Error handling policy	Human-readable error messages; critical incidents resolved within 24 hours, others within 48 hours.	Team
Compliance	Full GDPR adherence; HTTPS for all communication; no raw card data stored.	Team
Deployment & portability	Use Docker containers for reproducibility on Linux and Windows.	Cristiana
Future roadmap	Add build-your-own basket, discounts, and recommendations after MVP.	Cristiana

Conclusion

The meeting successfully established a shared vision and a realistic implementation plan for **BaskIT**. Under the leadership of **Cristiana-Florentina Precup** as Product Owner, the team defined clear priorities that balance business needs with technical feasibility.

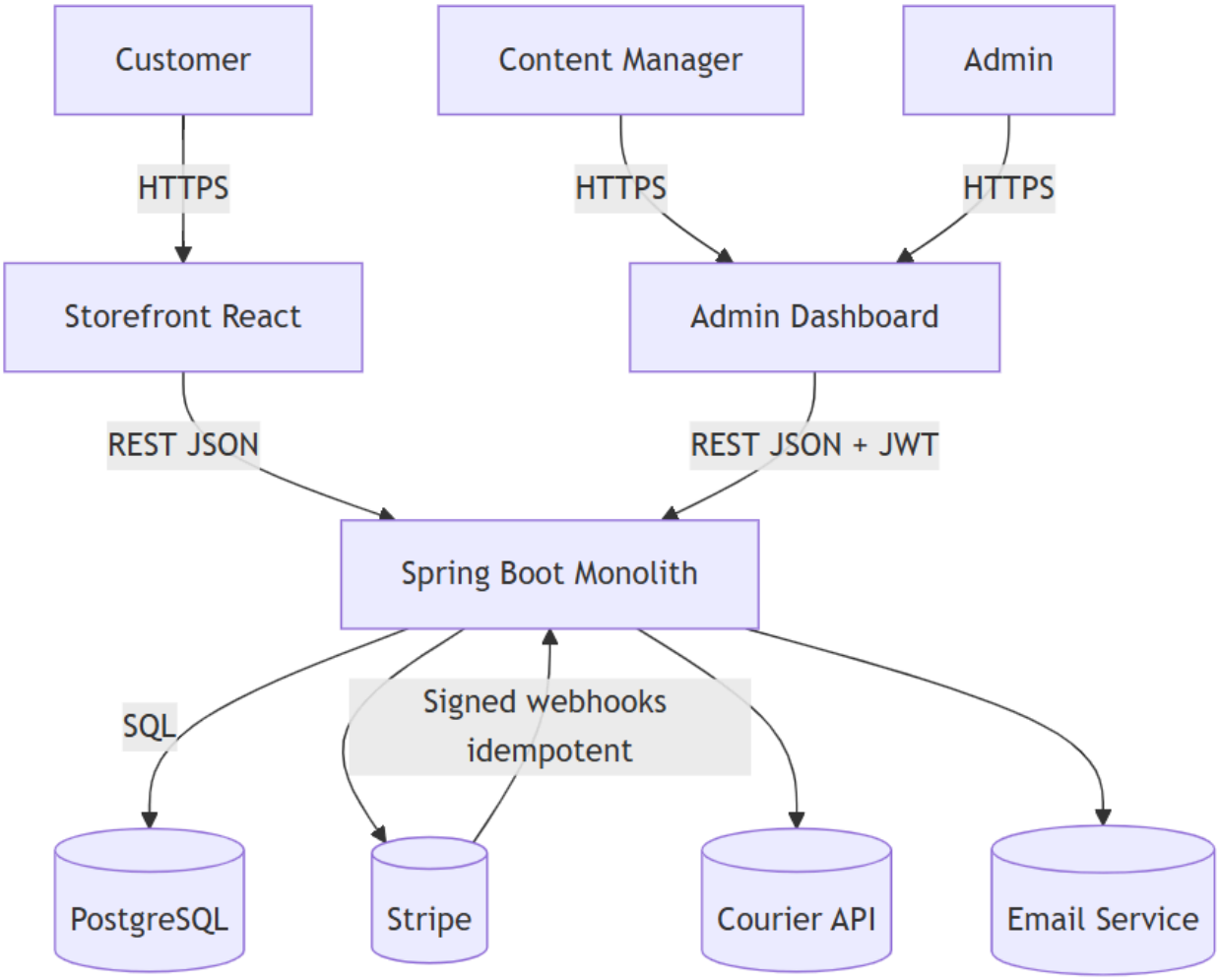
Mihai's architectural approach ensures reliability and maintainability, while Vlad’s UI focus guarantees accessibility and user satisfaction. Together, the team committed to building a stable and secure MVP emphasizing modularity, auditability, and extensibility.

The outcome of this meeting directly informs **Sections 2 and 3** of this SRS, ensuring every listed requirement, diagram, and constraint traces back to a consensus reached during this initial stakeholder session.

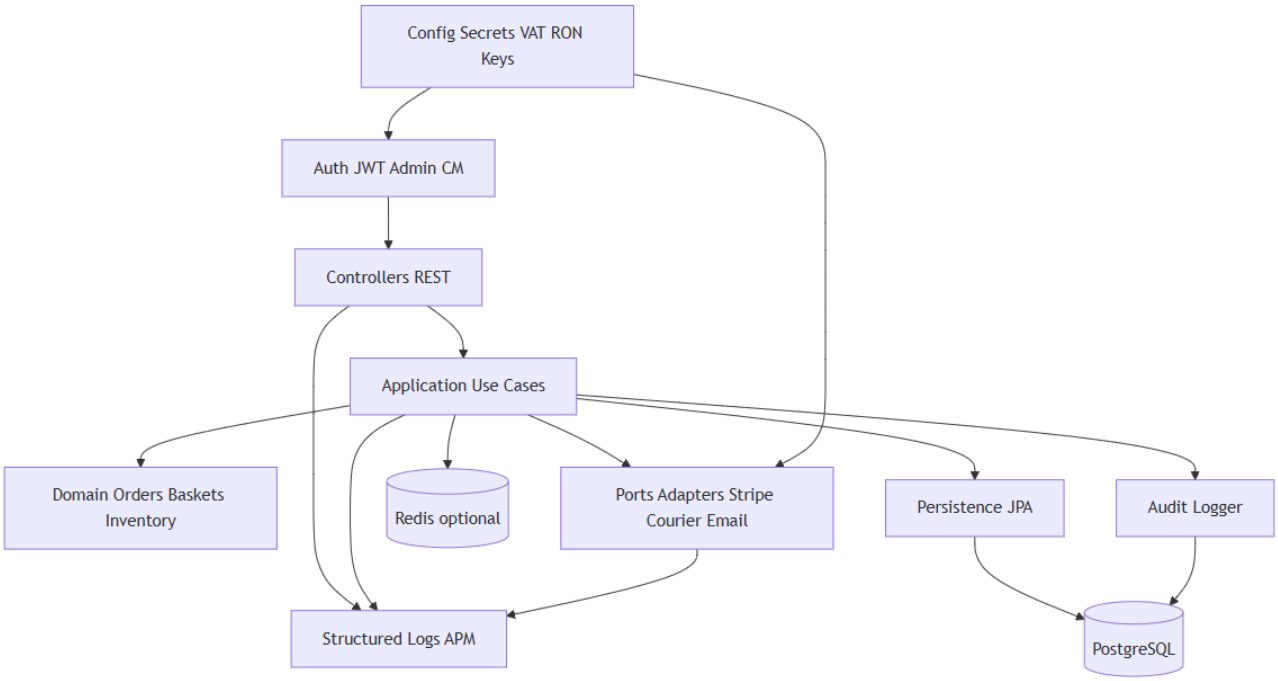
A2. System diagram

Description: A monolithic Spring Boot backend (hexagonal style) serves a React frontend. Integrates with Stripe (payments/webhooks), a courier adapter (labels/tracking), and an email provider. PostgreSQL stores core data.

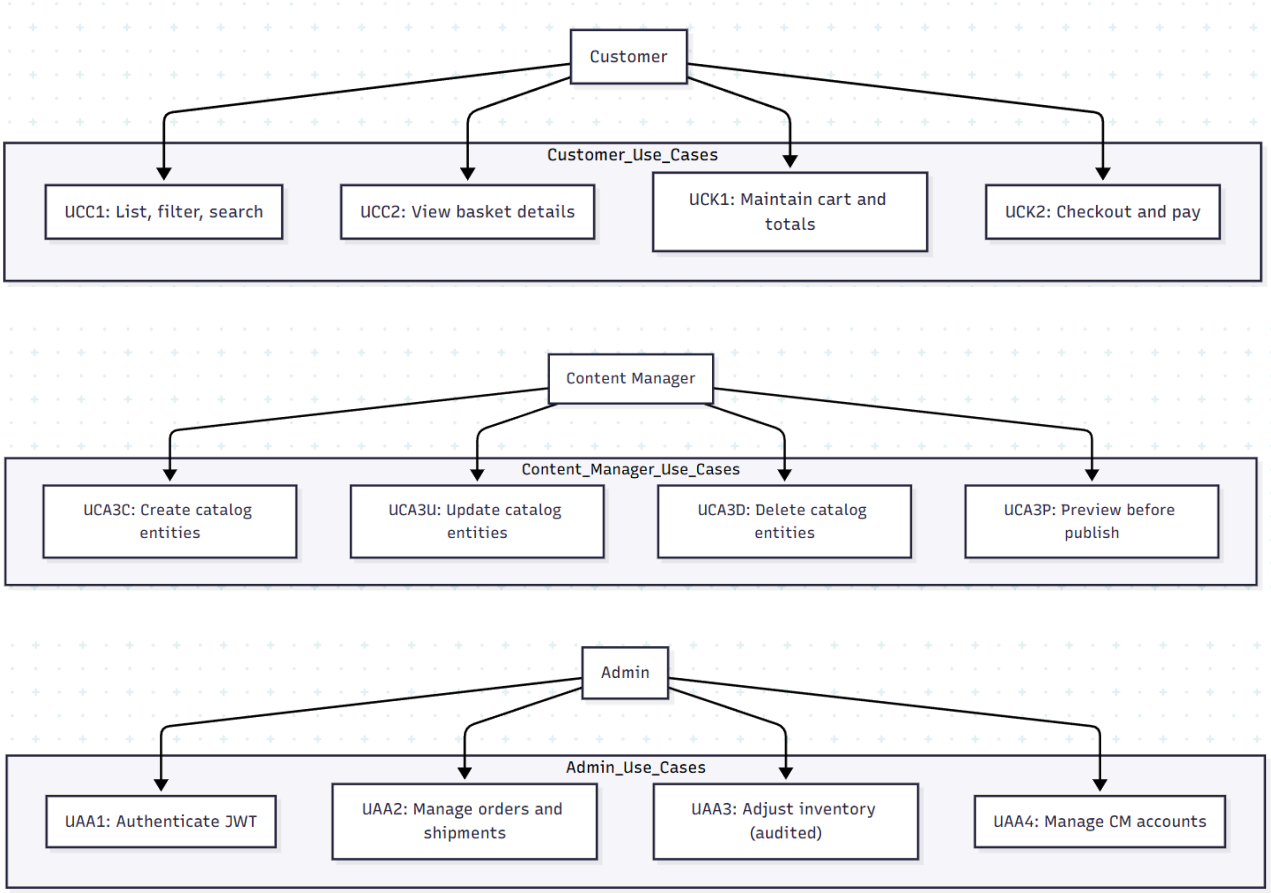
A2.1 Context



A2.2 Internal architecture (inside the monolith)



A3. Use Cases Diagrams



A4. Class Diagrams

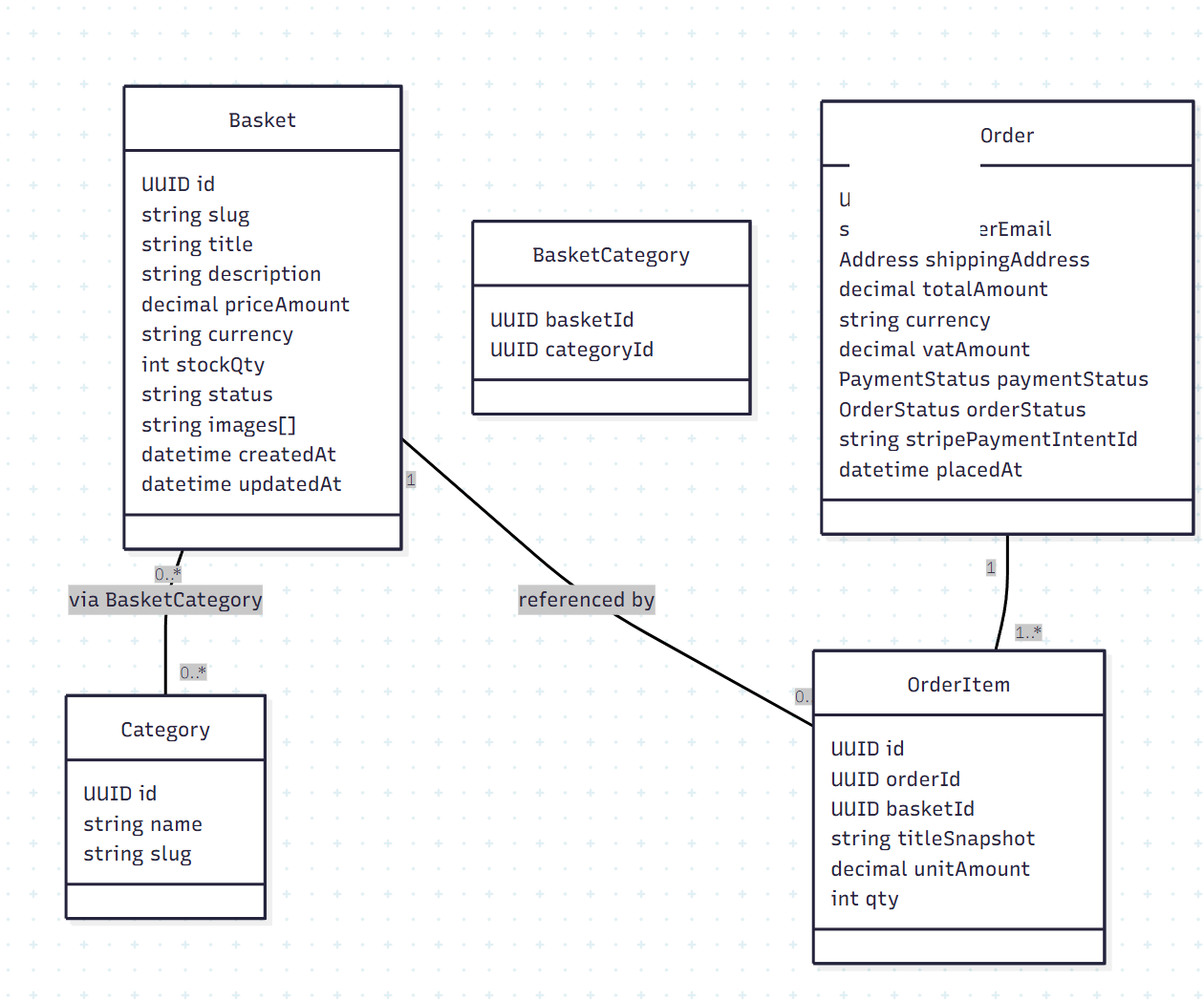
- **Basket** (id, slug, title, description, priceAmount, currency, stockQty, status, images[], createdAt, updatedAt)
- **Category** (id, name, slug)
- **BasketCategory** (basketId, categoryId) — join table for Basket ↔ Category (M:N)
- **Order** (id, customerEmail, shippingAddress: Address, totalAmount, currency, vatAmount, paymentStatus, orderStatus, stripePaymentIntentId, placedAt)
- **OrderItem** (id, orderId (FK), basketId (FK), titleSnapshot, unitAmount, qty) — 1..* from Order
- **Shipment** (id, orderId (unique FK), carrier, service, trackingNo, labelUrl, status) — 1:1 with Order
- **User** (id, email, passwordHash, role ∈ {ADMIN, CONTENT_MANAGER}, active, createdAt)
- **AuditLog** (id, actorUserId, actorEmail, actorIp, action, entityType, entityId, beforeJson, afterJson, occurredAt)
- **WebhookEvent** (id, provider, eventType, rawJson, receivedAt, processedAt)
- **Address** (fullName, line1, line2, city, county, postalCode, country, phone)

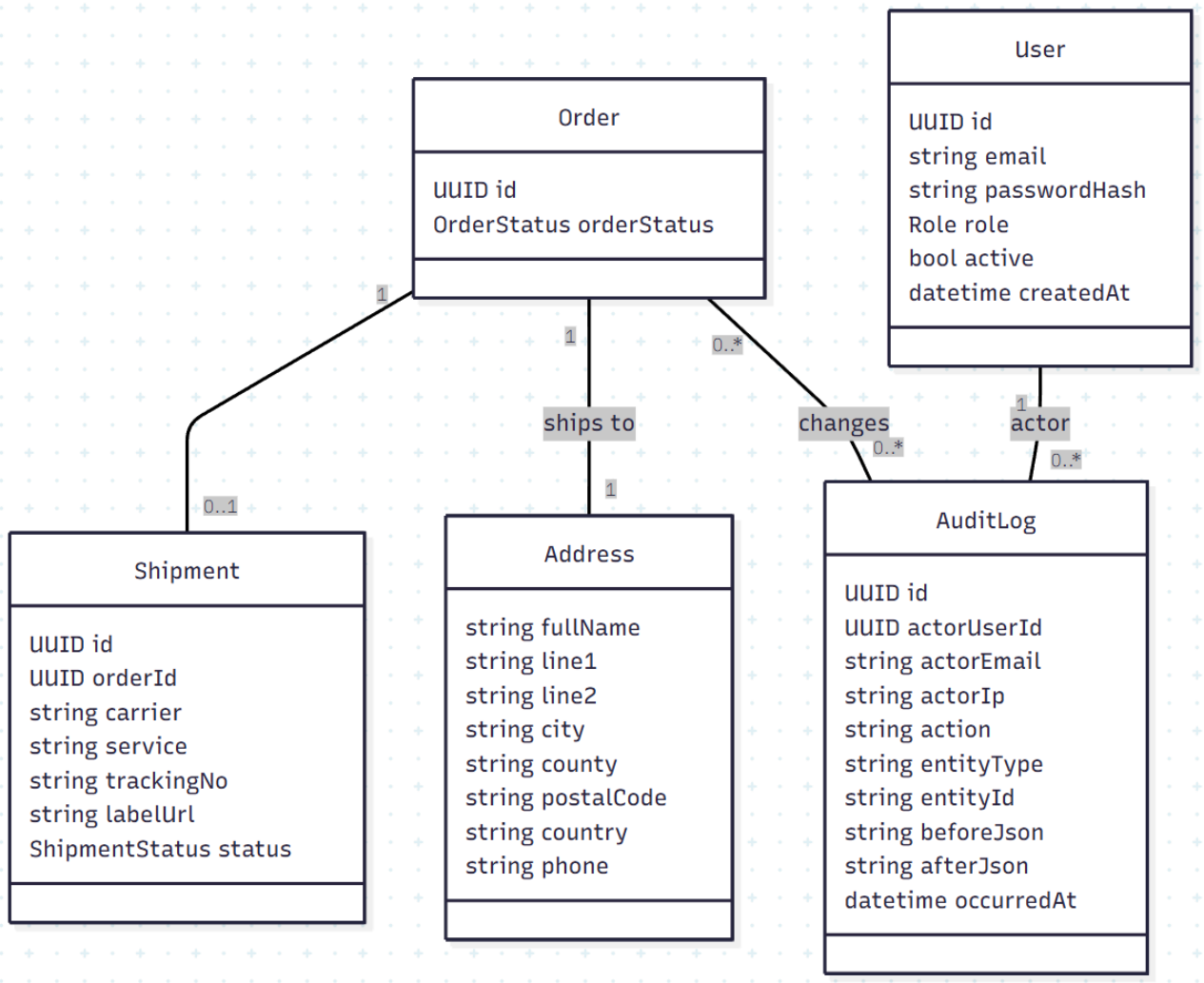
Relationships:

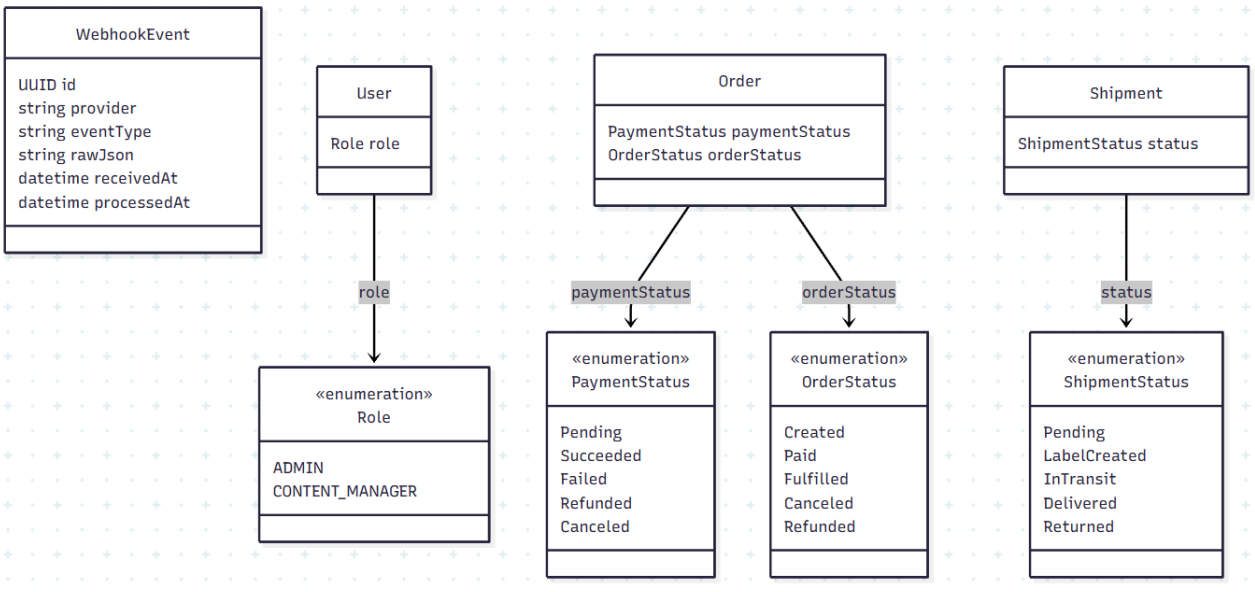
- **Basket** and **Category**: each Basket can belong to many Categories, and each Category can include many Baskets (implemented via the BasketCategory join table).
- **Order** and **OrderItem**: each Order contains one or more OrderItems.
- **Order** and **Shipment**: each Order has at most one Shipment.
- **Order** and **Address**: each Order ships to exactly one Address.
- **User** and **AuditLog**: a User can have zero or many AuditLog entries as the actor.
- **Order** and **AuditLog**: an Order can have zero or many AuditLog entries associated with its changes.

Enumerations:

- **Role** { ADMIN, CONTENT_MANAGER }
- **OrderStatus** { Created, Paid, Fulfilled, Canceled, Refunded }
- **PaymentStatus** { Pending, Succeeded, Failed, Refunded, Canceled }
- **ShipmentStatus** { Pending, LabelCreated, InTransit, Delivered, Returned }



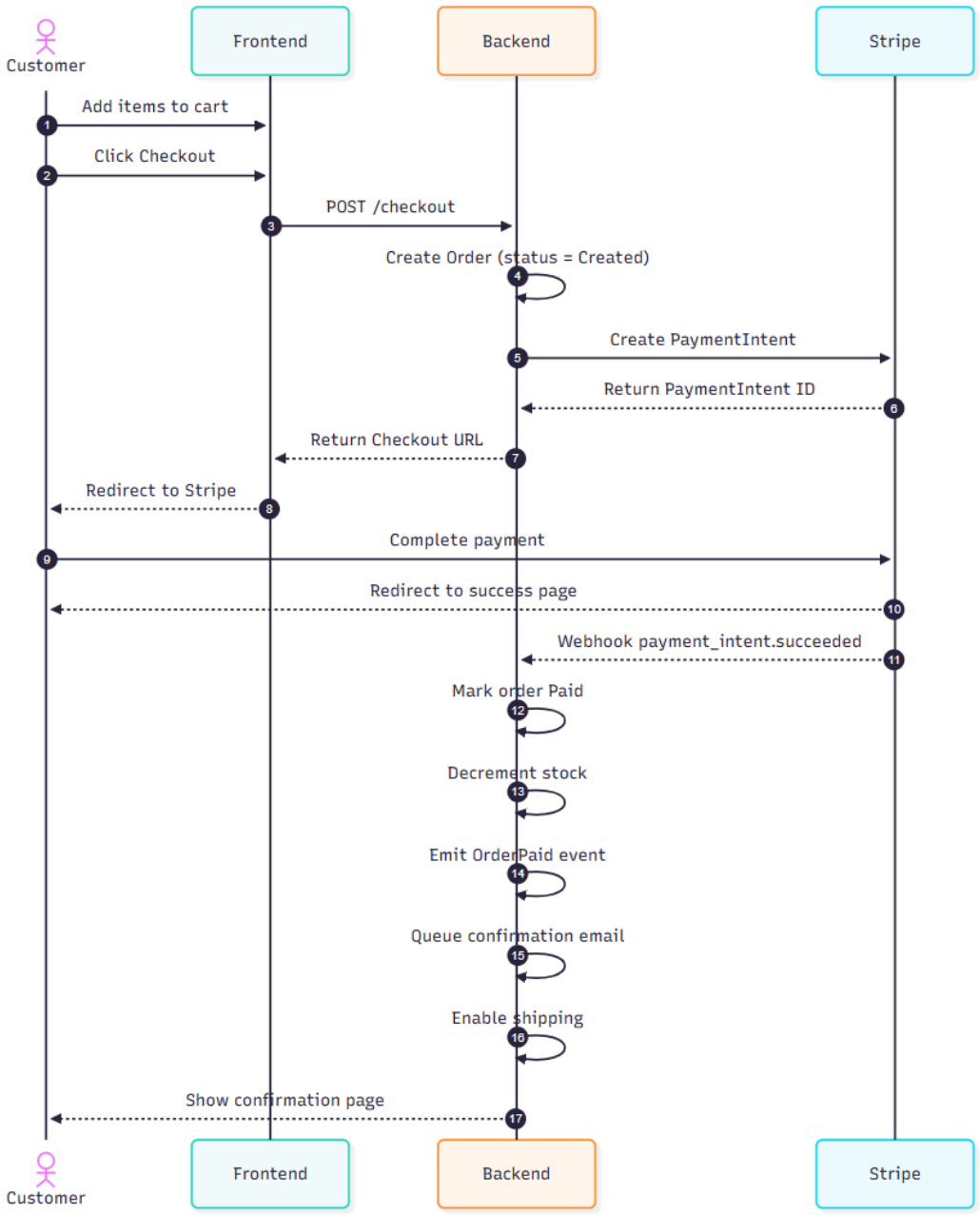

Order
UUID id
OrderStatus orderStatus
User
UUID id
string email
string passwordHash
Role role
bool active
datetime createdAt
1
1
0..*
ships to
changes
1
actor
0..*
0..*
0..1
Shipment
UUID id
UUID orderId
string carrier
string service
string trackingNo
string labelUrl
ShipmentStatus status
Address
string fullName
string line1
string line2
string city
string county
string postalCode
string country
string phone
AuditLog
UUID id
UUID actorUserId
string actorEmail
string actorIp
string action
string entityType
string entityId
string beforeJson
string afterJson
datetime occurredAt


WebhookEvent
UUID id
string provider
string eventType
string rawJson
datetime receivedAt
datetime processedAt
User
Role role
role
«enumeration»
Role
ADMIN
CONTENT_MANAGER
Order
PaymentStatus paymentStatus
OrderStatus orderStatus
paymentStatus
orderStatus
«enumeration»
PaymentStatus
Pending
Succeeded
Failed
Refunded
Canceled
«enumeration»
OrderStatus
Created
Paid
Fulfilled
Canceled
Refunded
Shipment
ShipmentStatus status
status
«enumeration»
ShipmentStatus
Pending
LabelCreated
InTransit
Delivered
Returned

A5. Sequence Diagrams

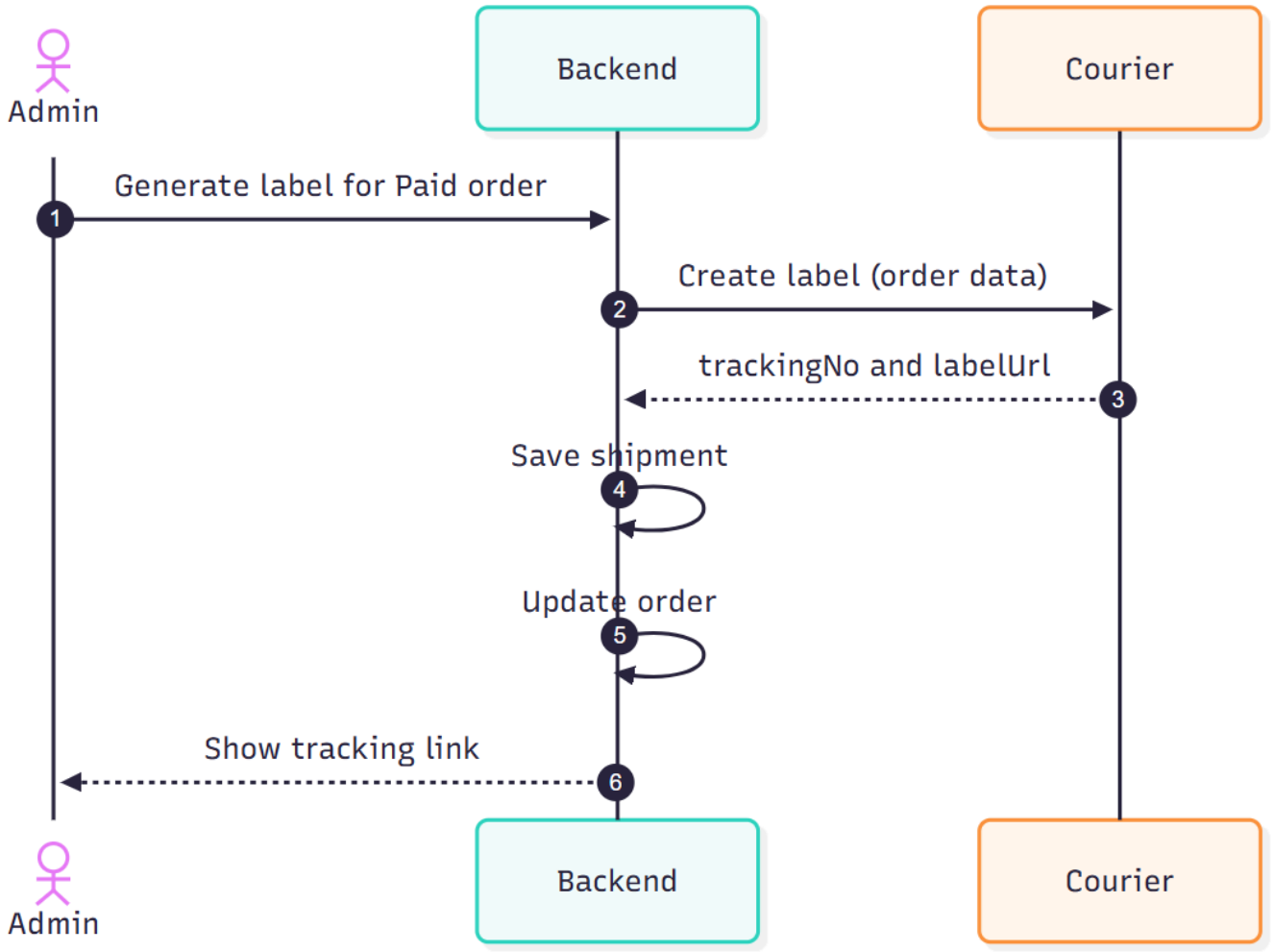
SD-1: Checkout & Pay

1. Customer adds items to cart.
2. Customer submits checkout → **Backend** creates Order{status=Created}, creates Stripe Payment Intent, returns Checkout URL.
3. Customer pays on Stripe → Stripe redirects to success page.
4. Stripe sends payment_intent.succeeded webhook → Backend marks order Paid, decrements stock, emits OrderPaid event → email confirmation queued, shipping enabled.



SD-2: Generate Shipping Label (admin)

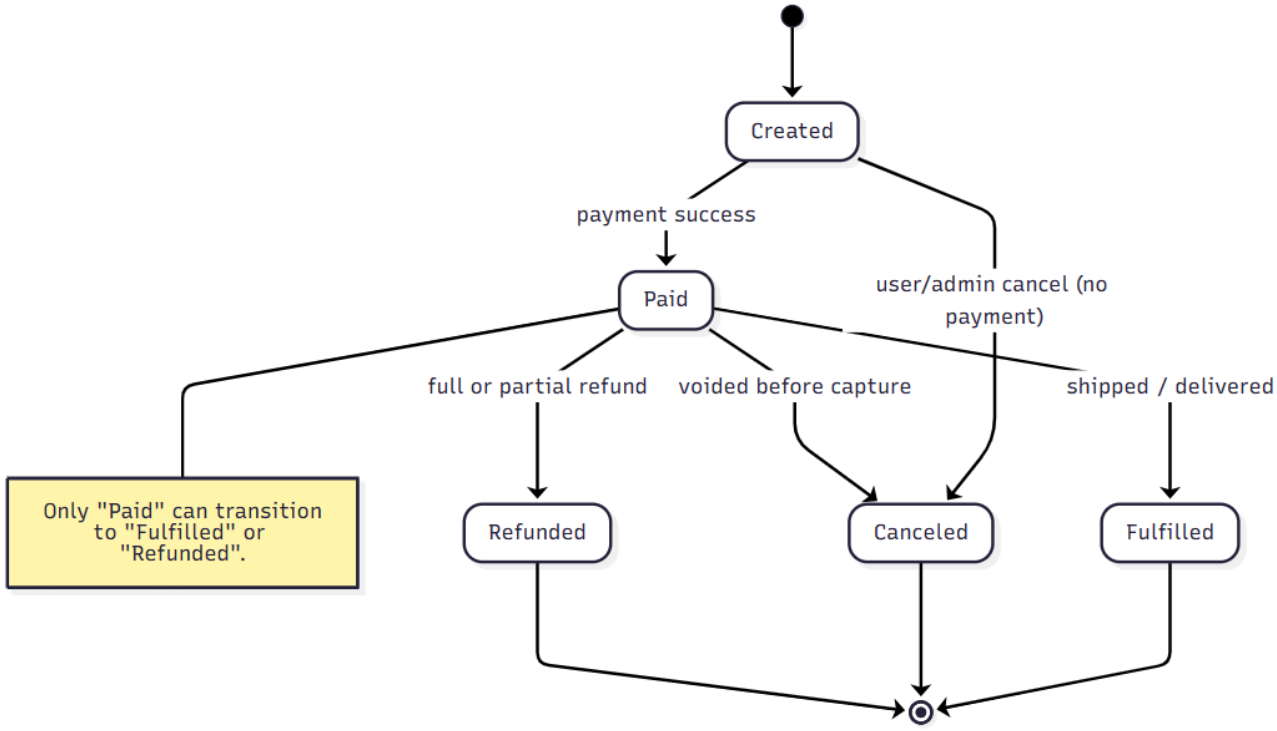
1. Admin opens a Paid order → clicks “Generate Label”.
2. Backend validates order is Paid and not yet shipped → calls courier adapter to create label.
3. Courier returns tracking number and label URL.
4. Backend saves Shipment, logs audit entry, and exposes tracking on the order.
5. Admin sees the tracking link; customer order page now shows tracking info.



A6. State Diagrams

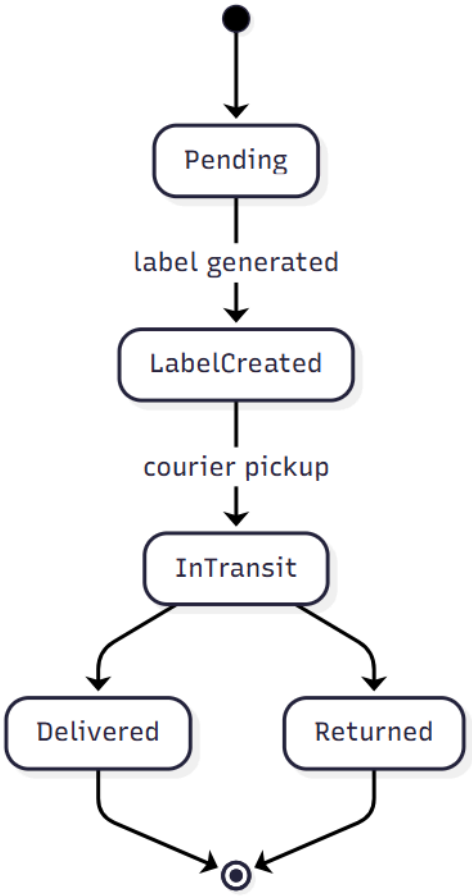
Order state machine

- **States:** Created, Paid, Fulfilled, Canceled, Refunded
- **Transitions:**
  - Created → Paid (payment success)
  - Created → Canceled (user/admin cancel; no payment)
  - Paid → Fulfilled (shipped/delivered)
  - Paid → Refunded (full/partial refund)
  - Paid → Canceled (only if payment voided before capture)
- **Guards:** Only Paid can become Fulfilled or Refunded. Audit entry on each transition.

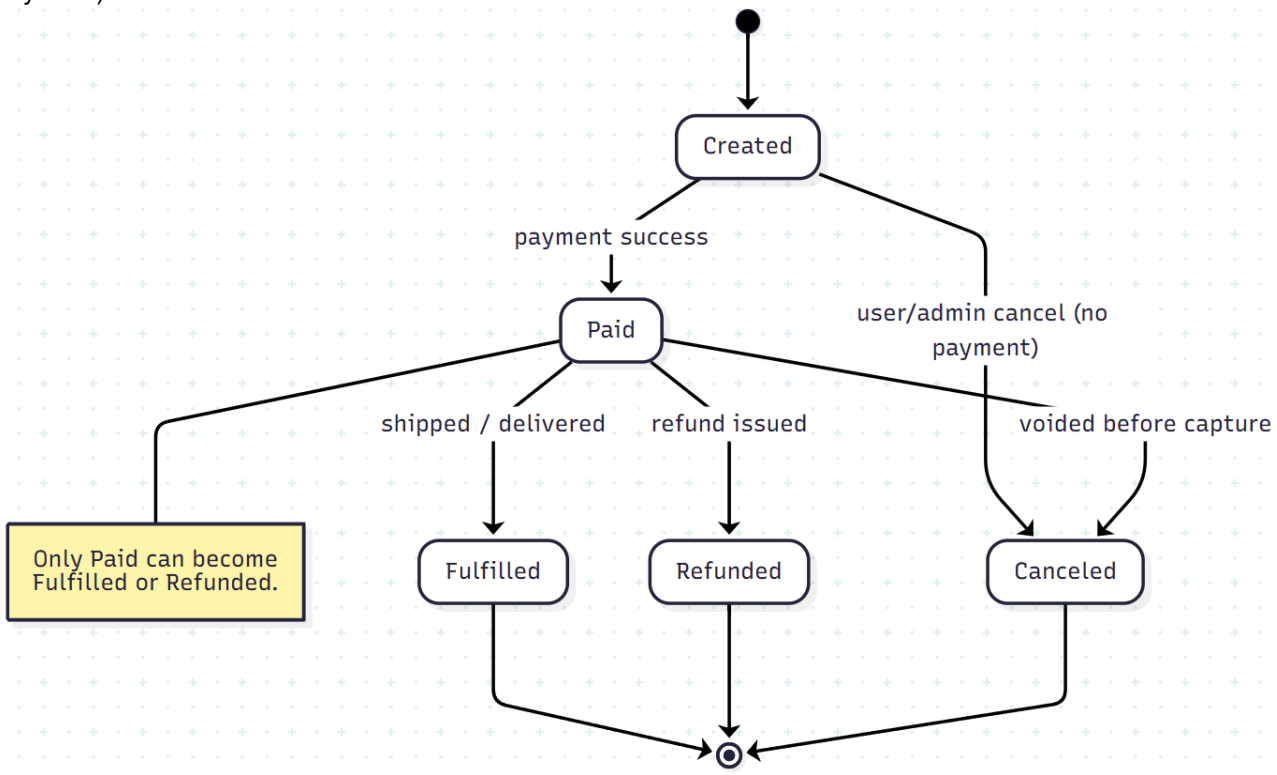


Shipment state

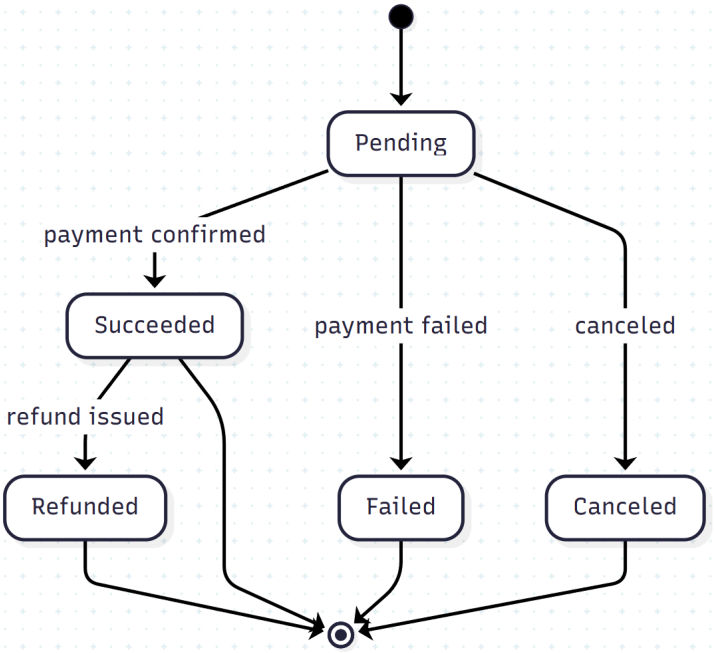
- Pending → LabelCreated → InTransit → Delivered | Returned (based on carrier updates).



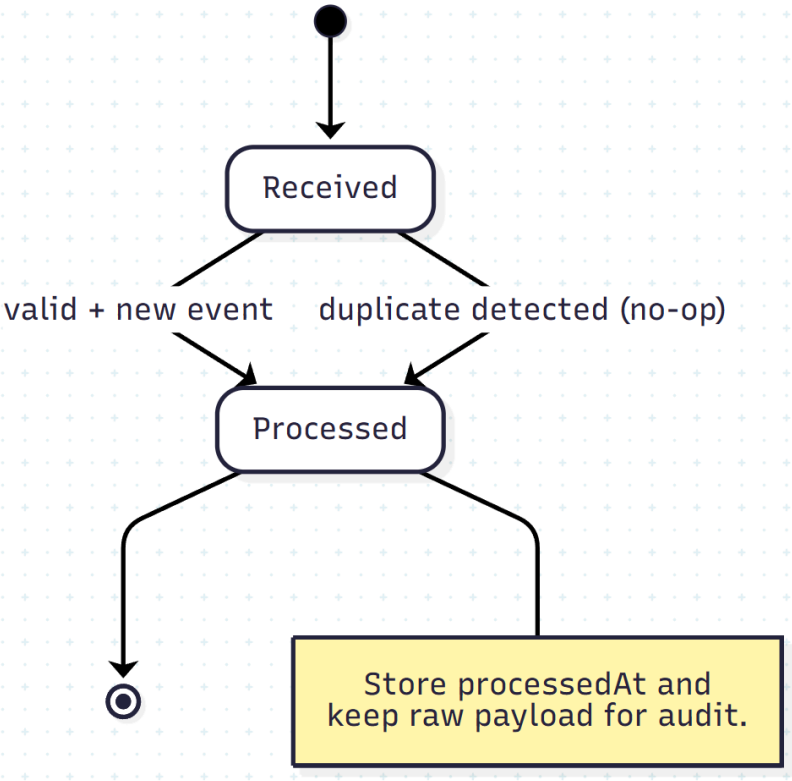
Order state
Created → Paid → Fulfilled | Refunded | Canceled (voided before capture); Created → Canceled (no payment).



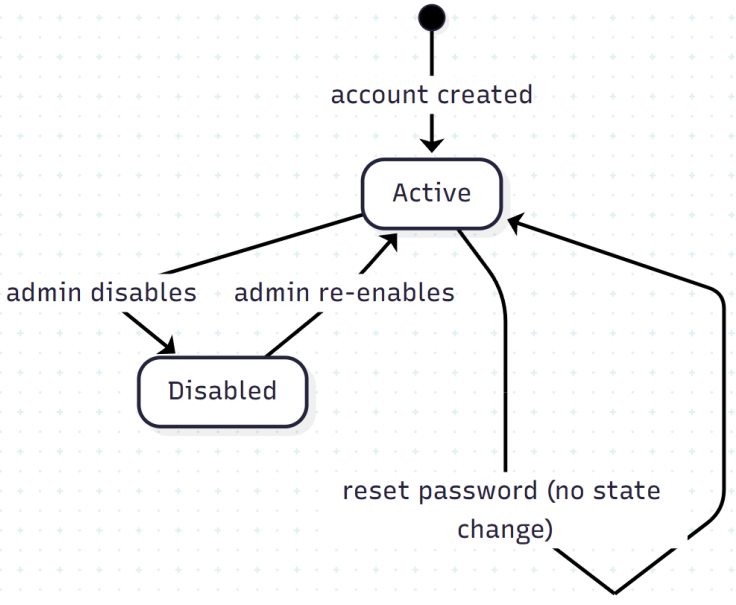
Payment status
Pending → Succeeded | Failed | Canceled; Succeeded → Refunded.



Webhook event processing
Received → Processed (valid, first delivery) | Processed (duplicate, idempotent).



User account state
Active ↔ Disabled (admin toggle); Reset password: no state change.



A7. Document Evolution

Version	Date	Author(s)	Changes
1.0	2025-10-18	Cristiana Precup, Mihai Prodan, Vlad Dumitru	Initial SRS (Sections 1–3 + Appendices A1–A9 skeleton).
1.1	2025-10-20	Cristiana Precup, Vlad Dumitru, Mihai Prodan	Finalize SRS; agree on appendices content
1.2	*TBD*	*TBD*	*TBD*

A8. Report regarding team meetings

Meeting 1

- **Date/Location:** 2025-10-18, online
- **Participants:** Cristiana, Mihai, Vlad
- **Agenda:** Pick theme; confirm stack; outline MVP.
- **Decisions:** Use React + Spring Boot + PostgreSQL; per-basket inventory; Stripe; pluggable courier.
- **Action items:**
    - Cristiana: outline SRS Section 1
    - Mihai: outline SRS Section 2
    - Vlad: outline SRS Section 3.

Meeting 2

- **Date/Location:** 2025-10-20, campus
- **Participants:** Cristiana, Mihai, Vlad
- **Agenda:** Finalize SRS Sections 1–3; agree on appendices content.
- **Decisions:** Keep monolith with hexagonal style; adopt Strategy/State/Adapter/Observer.
- **Action items:**
    - Mihai: finalize Functional/Performance requirements.
    - Vlad: prepare diagrams handling details.
    - Cristiana: prepare diagrams webhook details.

A9. Conclusions regarding the activity

- The team defined a clear MVP aligned with course requirements and local constraints (RON, VAT, Romania).
- Risks are contained by using Stripe for PCI scope and a pluggable courier adapter to defer provider specifics.
- The chosen patterns (Strategy, State, Adapter, Observer) map directly to core problems (catalog logic, order lifecycle, integrations, side-effects) and will simplify testing and future extensions.
- Next steps: finalize provider selection for shipping, refine non-functional targets with basic load tests, and implement the payment → webhook → stock flow end-to-end.