



Master Big Data & Internet of Things

Academic Year 2025-2026

CompTek

A Moroccan scam-resistant marketplace for buying and selling game accounts

Prepared by:

Amine HAMOUCHI
Bendahmane GHOUFRANE

Supervising Professors:

El Faquih Loubna
Hirchoua Badr
Sekkate Sara

April 2026

Abstract

Secondary markets for video game accounts are common, but they are difficult to trust: buyers and sellers rarely know each other, payments and account handover are easy to dispute, and fraudulent behavior (mass listings, failed payments, suspicious reviews) is hard to spot without dedicated tooling. CompTek addresses this problem by providing an integrated marketplace platform where users can publish listings, negotiate, and complete trades through a controlled, auditable workflow rather than informal peer-to-peer deals.

CompTek is a marketplace for video game accounts that tackles trust and fraud in peer-to-peer trading. Instead of unregulated exchanges, the platform offers listings, buyer-seller trades with a strict status workflow, encrypted credential delivery, real-time trade chat, and admin moderation. An optional Big Data stack consumes marketplace events via Kafka, scores seller risk with a Python processor, and stores results in HBase/HDFS for fast lookup and historical analysis. The system is built with Spring Boot and PostgreSQL (Supabase) on the server, React and TypeScript on the client, and Docker-orchestrated Kafka, Hadoop, Hive, and HBase for analytics and fraud detection.

Mots-clés : marketplace, video game accounts, Spring Boot, React, PostgreSQL, Kafka, fraud detection, risk scoring, WebSocket, microservices, Docker, escrow workflow.

Acknowledgments

We would like to express our sincere gratitude to our supervising professors, Prof. Loubna El Faquih, Prof. Badr Hirchoua, and Prof. Sara Sekkate, for their guidance, availability, and valuable feedback throughout the development of this project.

Our thanks also extend to the faculty and staff of the Master Big Data & Internet of Things program at ENSAM for the quality of teaching and for providing an environment that encouraged us to explore modern data and software engineering practices.

Finally, we owe special thanks to our families and loved ones for their patience, encouragement, and unwavering support during this academic year.

Amine Hamouchi
Bendahmane Ghoufrane

Contents

Abstract	i
Acknowledgments	ii
1 General Introduction	1
1.1 General Context	1
1.1.1 Digital marketplaces and trust	1
1.1.2 Local and academic context	1
1.1.3 Technical landscape	2
1.2 Objectives	3
1.2.1 Functional objectives	3
1.2.2 Technical objectives	3
1.2.3 Non-objectives (current version)	3
1.3 Organization of the Report	4
2 Fundamental Concepts	5
2.1 Marketplace and Trade Domain	5
2.2 Java Enterprise Edition and Spring Boot	6
2.3 REST, DTOs, and Stateless APIs	6
2.4 Real-Time Messaging (WebSocket and STOMP)	6
2.5 Single-Page Front End (React)	6
2.6 Persistence and Schema Management	7
2.7 Event-Driven Processing and Big Data Building Blocks	7
2.8 Microservices vs. Modular Monolith	7
3 JEE Application	8
3.1 Solution Overview	8
3.2 Backend Layering	8
3.3 Core Domain Model	9
3.3.1 Identity, Access and Profiles	9
3.3.2 Marketplace Catalog	10
3.3.3 Trades, Delivery and Reviews	11
3.4 Trade State Machine	12
3.5 Marketplace and Listings	12
3.6 Administration Module	13
3.7 React Client (Presentation Layer)	13
3.8 Persistence and Migrations	16
3.9 Integration Points for Big Data	16
3.10 Chapter Conclusion	16

4	Application Security	17
4.1	Threat Model (Simplified)	17
4.2	Authentication	17
4.2.1	Email and password	17
4.2.2	JWT session model	18
4.2.3	Supabase bridge	19
4.3	Authorization (RBAC)	19
4.4	WebSocket Security	20
4.5	Protection of Delivered Credentials	20
4.6	Data Validation and Error Handling	21
4.7	Kafka and Internal APIs	22
4.8	Security Configuration Summary	23
5	Big Data Pipeline	24
5.1	Motivation	24
5.2	Architecture	24
5.3	Event Contract	26
5.4	Rule-Based Risk Scoring (MVP)	26
5.4.1	Score calculation (<code>scoring.py</code>)	26
5.4.2	Exposing the score to CompTek (<code>risk_api.py</code>)	27
5.5	Storage Layers	28
5.5.1	HDFS — data lake (infrastructure ready, sink planned)	28
5.5.2	Hive — SQL analytics on HDFS (infrastructure ready)	29
5.5.3	Summary	29
6	General Conclusion	30
6.1	Synthesis of the Work	30
6.2	Objectives Achieved and Remaining Gaps	31
6.3	Lessons Learned	31
6.4	Perspectives	32
6.5	Closing Remarks	32

List of Figures

1.1	CompTek system context: presentation, application, and data tiers.	2
2.1	Trade lifecycle state machine (simplified).	5
3.1	CompTek deployment and component overview.	8
3.2	JPA domain model: identity, roles, profiles, and seller payment setup.	10
3.3	JPA domain model: listings, categories, images, and game stats collection. . . .	11
3.4	JPA domain model: trade workflow, credentials, messages, and reviews.	12
3.5	Landing page (/).	13
3.6	Marketplace browse (/marketplace).	14
3.7	Sign-in (/login).	14
3.8	Listing detail (/listings/:id).	15
3.9	My trades (/trades).	15
3.10	Seller dashboard (/dashboard).	15
3.11	Create listing (/sell) with game metadata and media upload.	16
5.1	Big Data pipeline architecture.	25

List of Tables

4.1	RBAC aligned with <code>SecurityConfig</code>	19
4.2	Security mechanisms deployed in CompTek.	23
5.1	Fraud score rules (maximum theoretical score = 100).	26
5.2	Storage roles in the Big Data stack.	29

Chapter 1

General Introduction

This report presents *CompTek*, a full-stack software project developed in the framework of the Master Big Data & Internet of Things at ENSAM. The platform is a Moroccan-oriented marketplace for buying and selling video game accounts, designed to reduce scams and informal peer-to-peer risk through a structured trade workflow, application security, and an optional Big Data pipeline for seller risk scoring.

The work spans three repositories: a **Spring Boot** backend), a **React** web client , and a **Docker-based Big Data stack**. Together they illustrate how enterprise Java development, modern front-end engineering, and stream-oriented analytics can be combined in a single coherent product.

1.1 General Context

Secondary markets for video game accounts operate under a severe trust deficit due to opaque peer-to-peer handovers, rampant payment disputes, and a lack of transactional audit trails. In the Moroccan ecosystem, these challenges are amplified by limited access to global escrow services, pushing users toward risky, informal local arrangements denominated in Moroccan Dirhams (**MAD**). *CompTek* directly addresses this vulnerability by enforcing a strict, state-driven trade workflow that secures credentials and isolates intensive analytical profiling such as seller risk scoring from the core transactional platform.

1.1.1 Digital marketplaces and trust

- **Payment disputes:** the buyer pays but never receives valid credentials, or the seller delivers before payment is confirmed.
- **Opaque handover:** account identifiers are exchanged in plain text over insecure channels.
- **Fraud at scale:** mass listing creation, repeated failed payments, shared devices or IP addresses across accounts, and collusive review patterns are difficult to detect manually.
- **Weak traceability:** without audit logs and state machines, it is hard to reconstruct what happened during a dispute.

1.1.2 Local and academic context

In Morocco, many users prefer local payment methods and prices in **MAD** (Moroccan dirham). *CompTek* is positioned as a *scam-resistant* alternative: it does not replace a licensed payment

escrow provider in its current version, but it structures the exchange (listing, trade room, status transitions, encrypted delivery of credentials, and moderation hooks) so that both parties follow the same rules.

From an academic perspective, the project aligns with the Master Big Data & IoT curriculum: it requires building a robust **JEE application** (persistence, services, REST and WebSocket APIs), applying **application security** (authentication, authorization, cryptography), and implementing a **Big Data pipeline** (Kafka, distributed storage, near real-time processing) that consumes operational events without blocking the user journey.

1.1.3 Technical landscape

The solution follows a classic three-tier pattern:

1. **Presentation tier:** React 18 single-page application (Vite, TypeScript, Tailwind CSS), with TanStack Query for server state and STOMP over WebSocket for trade chat.
2. **Application tier:** Spring Boot 3.3 on Java 21—REST controllers, transactional services, Spring Security with JWT, Spring Kafka producer, and STOMP messaging.
3. **Data tier:** PostgreSQL hosted via Supabase as the system of record; optional Hadoop ecosystem (HDFS, Hive, HBase) and Kafka for analytics and fraud risk scoring.

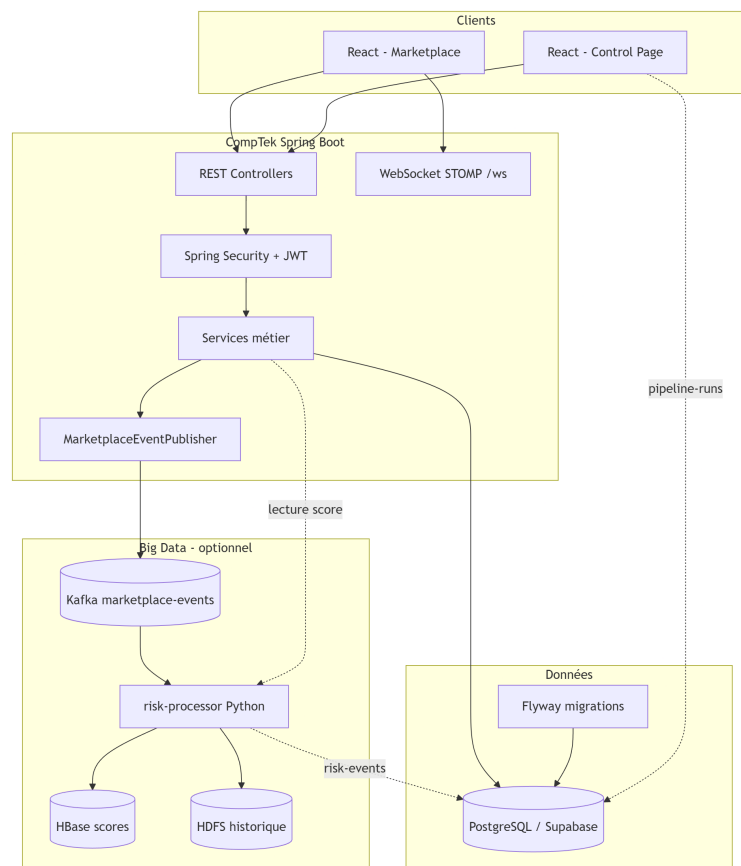


Figure 1.1: CompTek system context: presentation, application, and data tiers.

Feature flags (`KAFKA_PRODUCER_ENABLED`, `RISK_BRIDGE_ENABLED`) allow developers to run the marketplace locally without the full Big Data stack, which is an important pragmatic choice for iterative development and demonstrations.

1.2 Objectives

The project pursues both **functional** and **technical** objectives.

1.2.1 Functional objectives

- Provide a **marketplace** where sellers publish game account listings (categories, images, MAD pricing) and buyers discover offers.
- Implement a **guided trade flow** with an explicit state machine (CREATED → payment steps → delivery → COMPLETED), role-based transitions (buyer vs seller), and an audit trail (TradeEventLog).
- Offer **real-time communication** in a trade room via WebSocket/STOMP, using the same JWT as the REST API.
- Protect delivered account credentials with **AES-256-GCM** encryption at rest in the database.
- Support **user authentication** (email/password, Supabase/OAuth bridge, password reset) and profile management.
- Enable **administration** (user/listing/trade moderation, metrics, audit events) through secured `/admin/**` endpoints.
- Detect suspicious seller behavior through a **rule-based risk score** fed by marketplace events and exposed to the API via a risk bridge.

1.2.2 Technical objectives

- Apply a **layered JEE architecture** (controllers, DTOs, services, repositories, entities) with clear separation of concerns.
- Manage schema evolution with **Flyway** and validate the model with Hibernate (`ddl-auto=validate`).
- Design a **canonical event contract** (MarketplaceEvent) published to Kafka for downstream analytics.
- Deploy the analytics stack with **Docker Compose** (Kafka KRaft, HDFS, Hive, HBase, Python risk processor).
- Store historical events on **HDFS**, low-latency scores on **HBase**, and support batch-style reporting via **Hive**.
- Document architecture and implementation choices for reproducibility (this report and project docs/).

1.2.3 Non-objectives (current version)

The following items are explicitly out of scope for the delivered V1 product but may appear in future work: real financial escrow with a payment provider, multi-currency support beyond MAD-focused UX, full legal dispute resolution, and machine-learning-based fraud models (the MVP uses interpretable rules only).

1.3 Organization of the Report

After the abstract and acknowledgments, the body of this document is organized into four main chapters, followed by detailed design and data modeling sections where applicable.

Chapter 1 — Fundamental Concepts Introduces the domain (marketplace, trade, fraud), the main technologies (Spring ecosystem, React, Kafka, Hadoop components), and the architectural principles used in CompTek.

Chapter 2 — JEE Application Describes the Spring Boot backend and the React client: layering, main entities, trade lifecycle, persistence, APIs, and real-time features.

Chapter 3 — Application Security Covers authentication, JWT, role-based access control, credential encryption, WebSocket security, and integration with Supabase Auth.

Chapter 4 — Big Data Pipeline Presents the event-driven pipeline: Kafka topics, Python stream processor, HDFS/Hive/HBase roles, risk scoring rules, and linkage back to the Java API.

Subsequent sections of the report (project design, data sources, entity schemas, conclusions) complement these chapters with diagrams, tables, and forward-looking notes. The reader is encouraged to consult the repository documentation (`architecture-globale.md`, `docker-kafka-fraud-risk`) for implementation-level detail.

Chapter 2

Fundamental Concepts

This chapter establishes the vocabulary and technical background required to understand CompTek. It does not replace hands-on documentation; it frames how the concepts fit together in our project.

2.1 Marketplace and Trade Domain

A **listing** is an offer published by a seller (game title, rank, price in MAD, category, media). A **trade** is a bilateral transaction started from a listing: one buyer and one seller, one payment method, and a sequence of statuses that both parties advance according to their role. **Trade messages** form the chat history; some messages are system-generated when the status changes.

Trust is not only a UI concern: the backend enforces **allowed transitions** in an EnumMap inside TradeService, persists each change in TradeEventLog, and rejects illegal or unauthorized updates. This is closer to a workflow engine than a simple CRUD application.

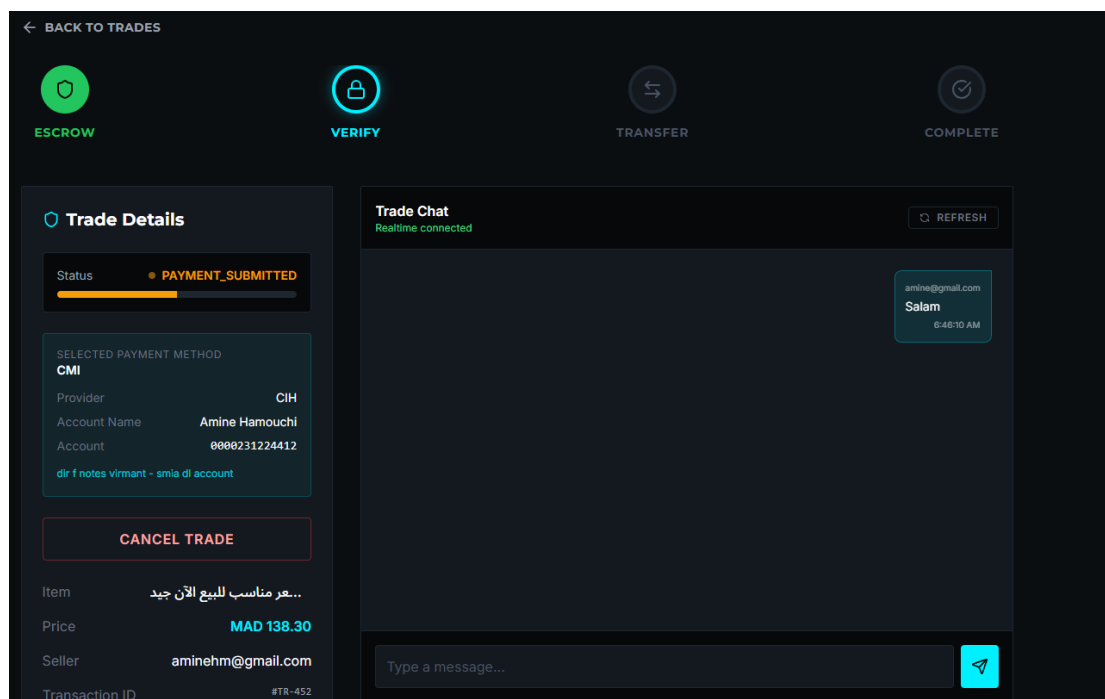


Figure 2.1: Trade lifecycle state machine (simplified).

2.2 Java Enterprise Edition and Spring Boot

Historically, **Java Enterprise Edition (JEE/Jakarta EE)** referred to a set of specifications for server-side Java: Servlets, JPA, CDI, JMS, etc. In practice, many modern projects—including CompTek—use **Spring Boot** as the productivity layer on top of these ideas:

- **Spring MVC / Spring Web** for REST endpoints.
- **Spring Data JPA** for ORM access to PostgreSQL.
- **Spring Security** for authentication and authorization filters.
- **Spring Kafka** for asynchronous event publication.
- **Spring WebSocket** with STOMP for messaging brokers.

CompTek targets **Java 21** and **Spring Boot 3.3.5**. The codebase follows a **layered architecture**: `controller` → `service` → `repository` → `entity`, with `dto` packages isolating API contracts from persistence.

2.3 REST, DTOs, and Stateless APIs

The backend exposes a **stateless REST API**: each request carries a JWT in the `Authorization` header. Controllers remain thin; validation uses Jakarta Bean Validation on request DTOs; responses use dedicated response DTOs so that JPA entities (lazy relations, internal fields) are never leaked.

Public read access is granted for marketplace discovery (`GET /listings/**`, `categories`). Mutations require an authenticated user with appropriate ownership or trade participation.

2.4 Real-Time Messaging (WebSocket and STOMP)

Trade chat uses **STOMP** over WebSocket at endpoint `/ws`. Clients subscribe to `/topic/trade/{id}`. The same `JwtService` that secures REST validates the token at connection time, keeping a single identity model for HTTP and messaging.

2.5 Single-Page Front End (React)

The React application (`React-CompTek-code-2`) is built with **Vite**, **TypeScript**, and **Tailwind CSS**. Key libraries include:

- **React Router** for routes (marketplace, listing detail, trades, trade room, dashboard, sell flow, settings).
- **TanStack Query** for caching and synchronizing REST data.
- **Supabase JS** for optional OAuth/sign-in flows bridged to CompTek JWT.
- **@stomp/stompjs** for trade room messaging.
- **i18next** for internationalization.

The front end is a **consumer** of the API; business rules for trade transitions remain on the server.

2.6 Persistence and Schema Management

PostgreSQL (via Supabase in deployment) is the **source of truth** for users, listings, trades, messages, payment methods, admin audit, and materialized pipeline run metadata.

Flyway applies versioned SQL migrations under `db/migration`. Hibernate is configured with `ddl-auto=validate`, so the ORM never auto-alter production schemas. This split is a recommended production practice.

2.7 Event-Driven Processing and Big Data Building Blocks

When the Big Data stack is enabled, the backend publishes **marketplace events** to Kafka topic `marketplace-events`. A canonical JSON schema (`docs/schemas/marketplace-event.schema.json`) defines fields such as `event_id`, `event_type`, `event_time`, `user_id`, and `metadata`.

Downstream components include:

- **Apache Kafka:** durable, ordered log of events; decouples OLTP (Spring) from analytics.
- **Python risk processor:** consumer group that updates rolling features and computes a score.
- **HDFS:** cheap storage for raw and processed datasets partitioned by date/hour.
- **Apache Hive:** SQL-like analytics on files in HDFS.
- **Apache HBase:** wide-column store for fast lookup of the latest risk score per seller entity.

If Kafka is unavailable, the publisher logs a warning and the user transaction still completes—a deliberate **resilience** choice.

2.8 Microservices vs. Modular Monolith

CompTek is logically split into three repositories but behaves as a **modular system**: the Spring application is a deployable monolith with clear packages; the Big Data stack is a separate compose project. This is lighter than full microservices operations (service mesh, distributed tracing everywhere) while still demonstrating integration across process boundaries (HTTP risk bridge, Kafka contracts).

Chapter 3

JEE Application

This chapter describes how the CompTek application is structured and how the main business capabilities are implemented in Spring Boot and React.

3.1 Solution Overview

Figure 3.1 summarizes the runtime architecture (clients, API, database, and optional Big Data). At runtime:

1. The user interacts with the React SPA.
2. The SPA calls REST endpoints on Spring Boot and opens a WebSocket for trade chat.
3. Services persist state in PostgreSQL and optionally emit Kafka events.
4. If enabled, `SellerRiskService` queries the Python risk API for seller scores.

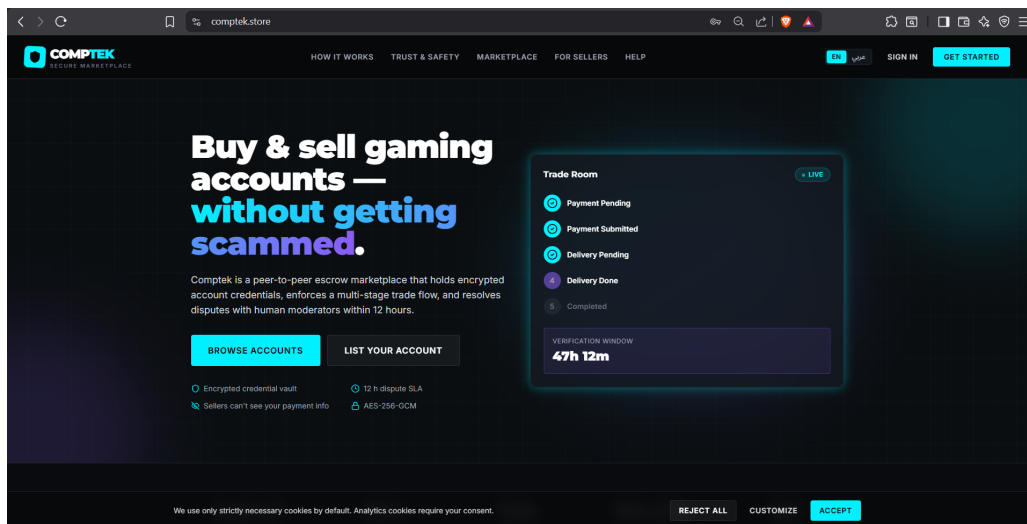


Figure 3.1: CompTek deployment and component overview.

3.2 Backend Layering

Package `com.example.Comptek` is organized as follows:

- `controller/` — REST resources (`AuthController`, `ListingController`, `TradeController`, `RiskController`, `admin/*`).
- `service/` — Business logic; `TradeService` owns the state machine; `admin/` for moderation use cases.
- `repository/` — Spring Data JPA interfaces.
- `entity/` — JPA mappings (`User`, `Listing`, `Trade`, `TradeMessage`, `profiles`, etc.).
- `dto/` — Request/response payloads.
- `security/`, `config/`, `exception/` — Cross-cutting concerns.

GlobalExceptionHandler centralizes API error responses. Configuration properties externalize secrets and feature flags via `application.properties` and optional `.env.properties`.

3.3 Core Domain Model

The platform defines three main actors (buyer, seller, administrator), mapped to roles `BUYER`, `SELLER`, and `ADMIN` on `User`. The JPA persistence model is split into three UML views (Figures 3.2–3.4), each covering a cohesive part of the data structure under `com.example.Comptek.entity`.

3.3.1 Identity, Access and Profiles

`User` is the authentication root: credentials, account flags, and a many-to-many link to `Role` via join table `user_roles`. Each user owns exactly one `Profile` (`@MapsId`: shared primary key). Optional specializations `BuyerProfile` and `SellerProfile` extend the profile with marketplace-specific attributes (e.g. `totalSpent`, `trustScore`, `verified`). Sellers configure payout instructions through `SellerBankDetails`, each row referencing a catalog `PaymentMethod`. Password recovery uses `PasswordResetToken`; Big Data job metadata is stored in `UserPipelineRun`. The profile stores `PreferredCurrency` (`USD`, `MAD`).

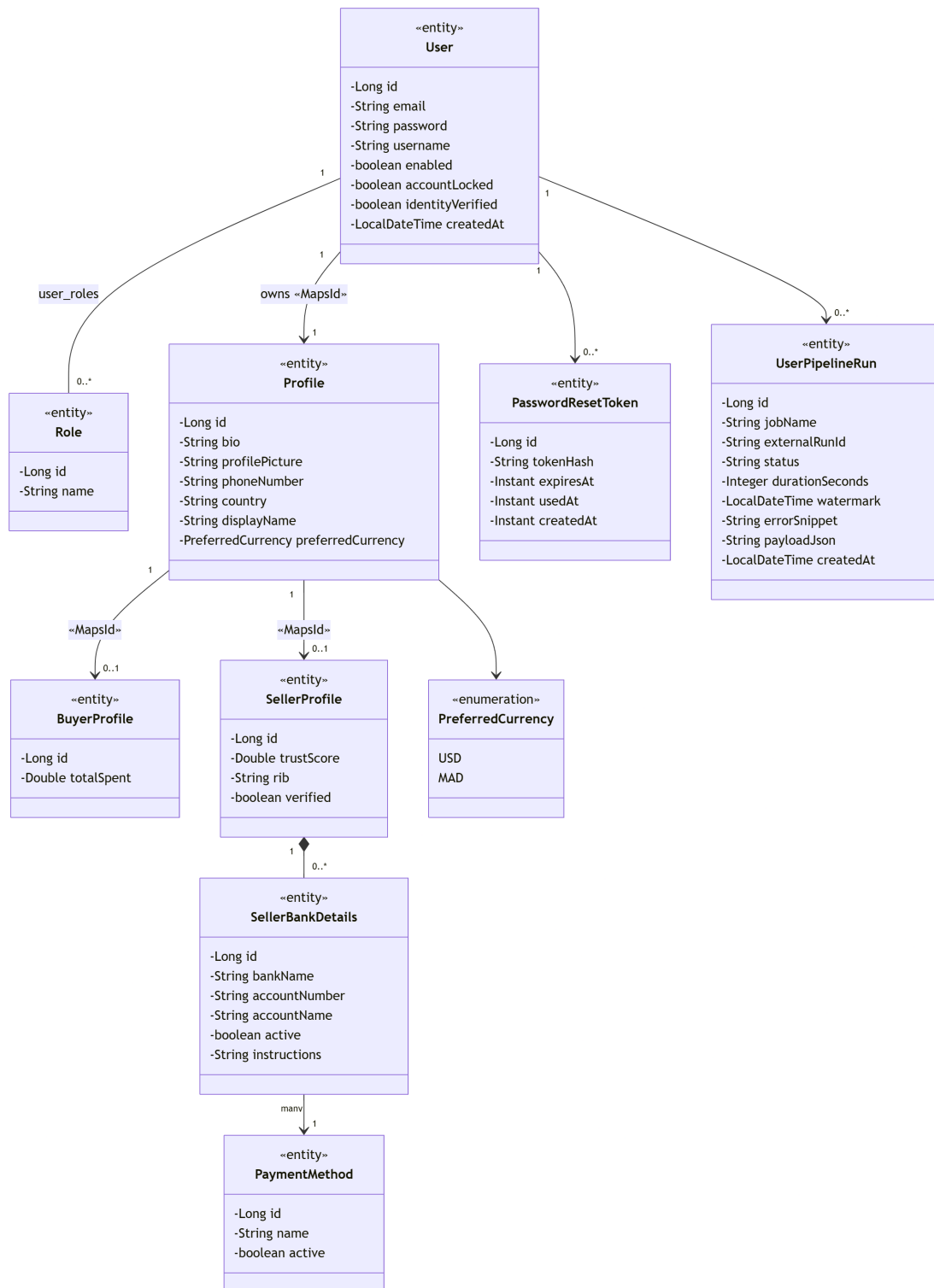


Figure 3.2: JPA domain model: identity, roles, profiles, and seller payment setup.

3.3.2 Marketplace Catalog

Listing is the sell-side aggregate: game metadata, price, moderation state (`ListingModerationStatus`), and a `@ManyToOne` link to the seller `User` and catalog `Category`. Images are modeled as `ListingImage` children (cascade ALL, ordered by `sortOrder`). Flexible per-game attributes are persisted as an `@ElementCollection` map (`listing_game_stats: stat_key / stat_value`), not as a separate entity class.

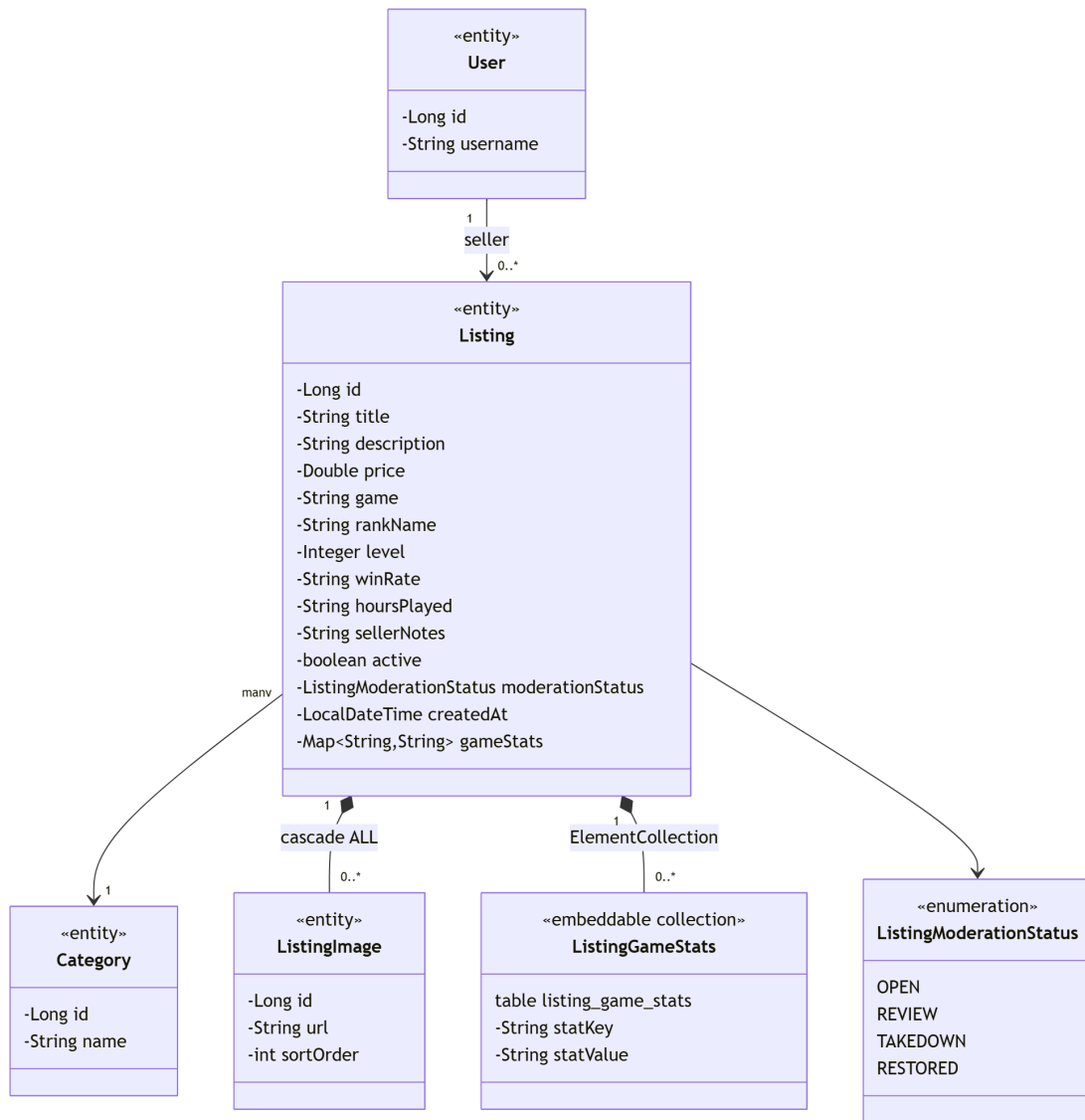


Figure 3.3: JPA domain model: listings, categories, images, and game stats collection.

3.3.3 Trades, Delivery and Reviews

Trade binds buyer, seller, and Listing, and records payment context (PaymentMethod, SellerBankDetails, plus snapshot columns for auditability). Status follows TradeStatus; each transition can be traced in TradeEventLog (TradeEventType, optional fromStatus/toStatus, actor). Account credentials are stored encrypted in TradeCredential (one-to-one with the trade). TradeMessage supports buyer-seller chat; Review links reviewer, reviewed user, and trade. Notification and AdminAuditEvent are documented separately (not shown in this trade view).

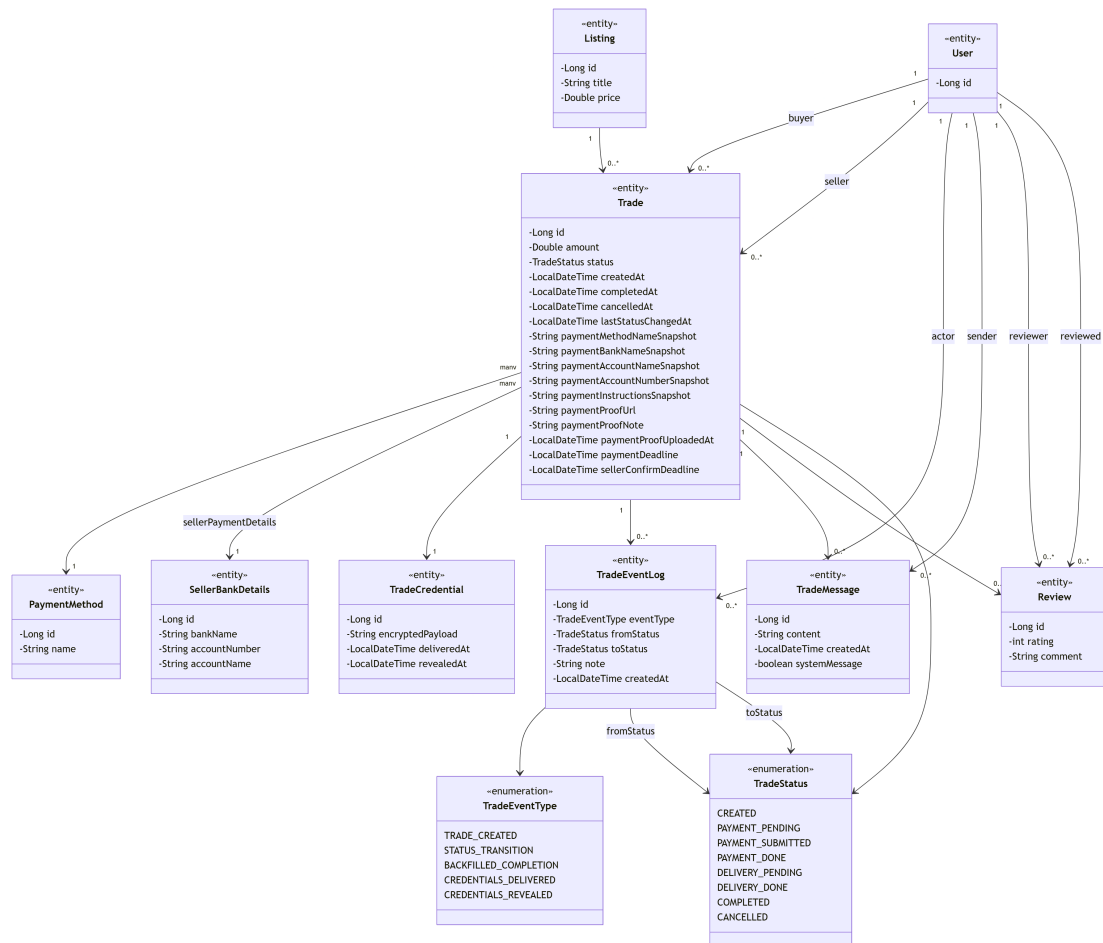


Figure 3.4: JPA domain model: trade workflow, credentials, messages, and reviews.

3.4 Trade State Machine

The workflow in Figure 2.1 is enforced in code by `TradeService`.

The trade lifecycle is the heart of the application. Implemented statuses include `CREATED`, `PAYMENT_PENDING`, `PAYMENT_SUBMITTED`, `PAYMENT_DONE`, `DELIVERY_PENDING`, `DELIVERY_DONE`, `COMPLETED`, and `CANCELLED` from several states.

`TradeService` maintains `ALLOWED_TRANSITIONS`: only valid edges are permitted. Each successful transition appends a `TradeEventLog` record. SQL check constraints (Flyway) align the database with application rules.

Buyers and sellers invoke different transition endpoints; admins can override certain operations for dispute resolution.

3.5 Marketplace and Listings

`ListingService` handles creation, update, and retirement of listings. Public endpoints support search and detail views. Listing mutations can emit `listing_created` and `listing_updated` events when Kafka is enabled—inputs for fraud features (velocity of new listings).

3.6 Administration Module

Administrative APIs under `/admin/**` require the `ADMIN` role. Capabilities include user suspension, listing takedown, trade intervention, platform metrics, and audit log consultation (`AdminAuditService`). Pipeline run history per user can be read from materialized table `user_pipeline_run` when populated by downstream jobs.

3.7 React Client (Presentation Layer)

The web client (`React-Comptek-code-2`, folder `fancy marketplace ui`) is a single-page application built with **Vite**, **TypeScript**, and **Tailwind CSS**. Routing is declared in `AppRoutes.tsx`: public landing and authentication; protected marketplace, trades, trade room (`/trades/:tradeId`), seller dashboard, listing creation, inventory, settings, and withdraw flows.

AuthContext persists the CompTek JWT (and optional Supabase OAuth) in `localStorage`; `ProtectedRoute` guards seller and buyer pages. **TanStack Query** loads listing and trade data from the Spring API; **STOMP** over `/ws` powers real-time trade chat. The interface uses glass-morphism cards, neon accents, English/Arabic i18n, and responsive layouts suited to a gaming marketplace.

Figures 3.5–3.11 show representative screens captured from the running application (Vite dev server on port 5173, API on port 8080, demo seller account).

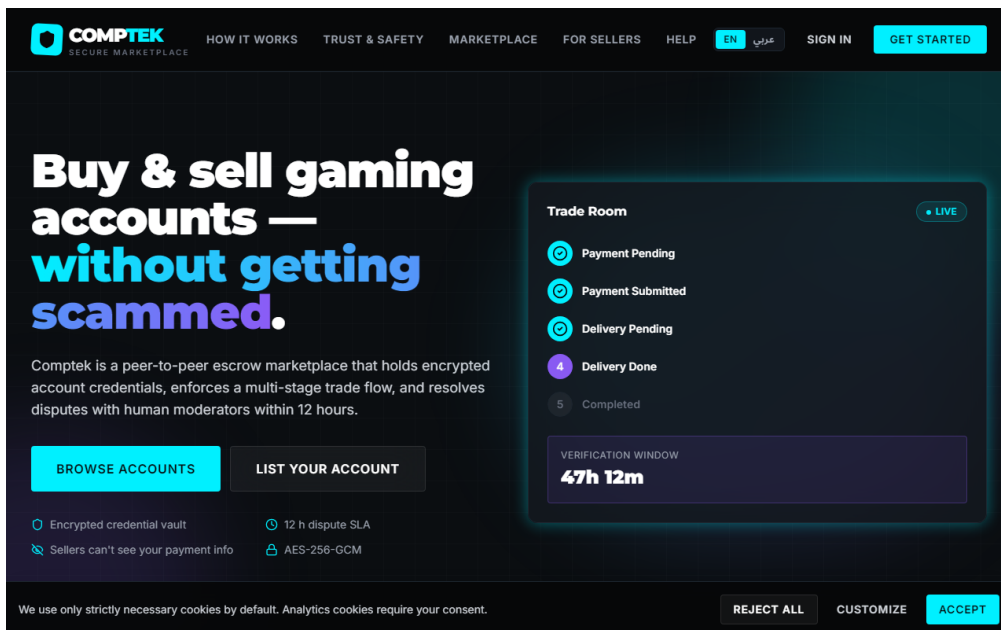


Figure 3.5: Landing page (/).

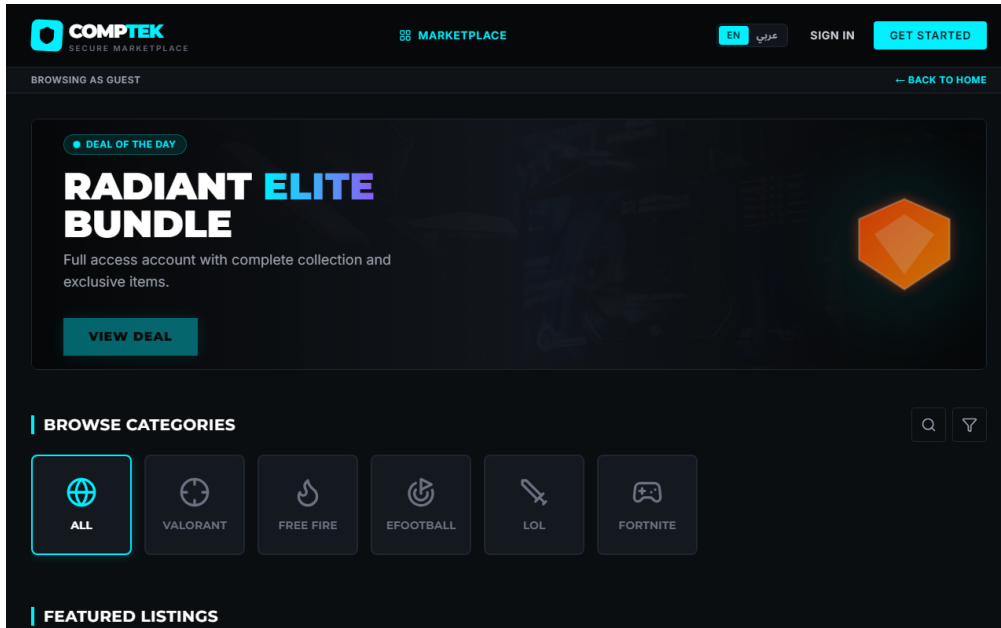


Figure 3.6: Marketplace browse (/marketplace).

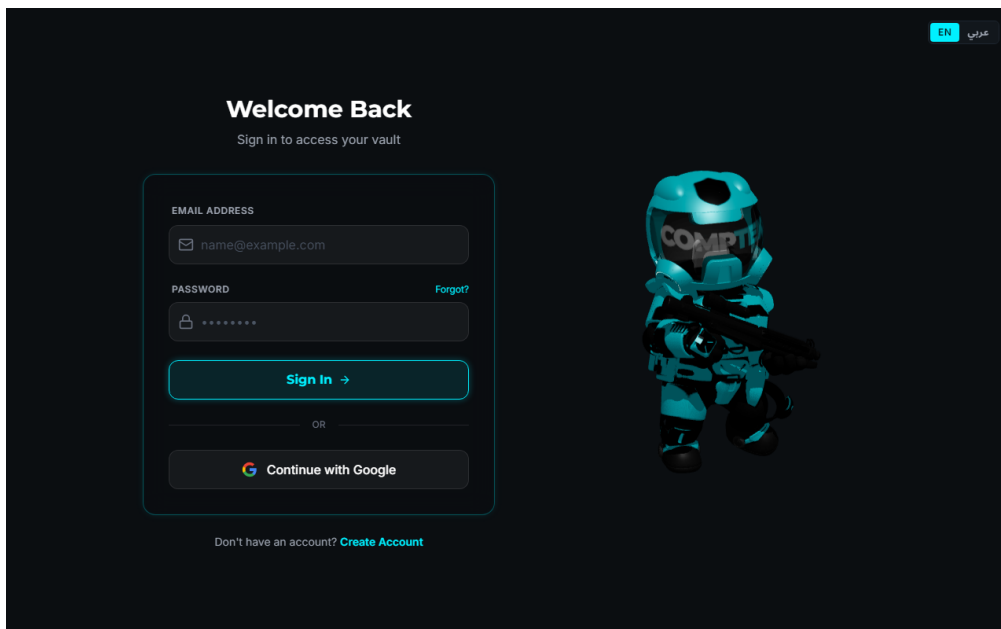


Figure 3.7: Sign-in (/login).

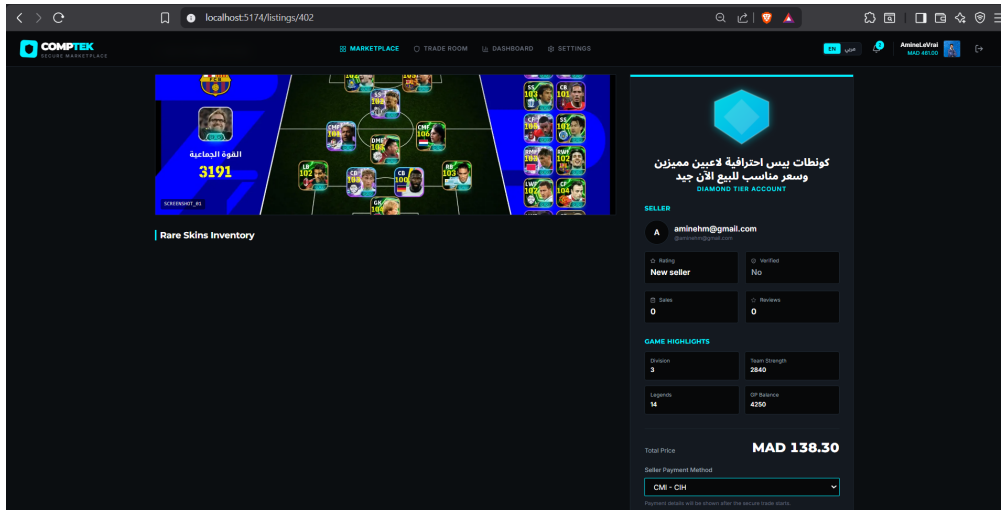


Figure 3.8: Listing detail (/listings/:id).

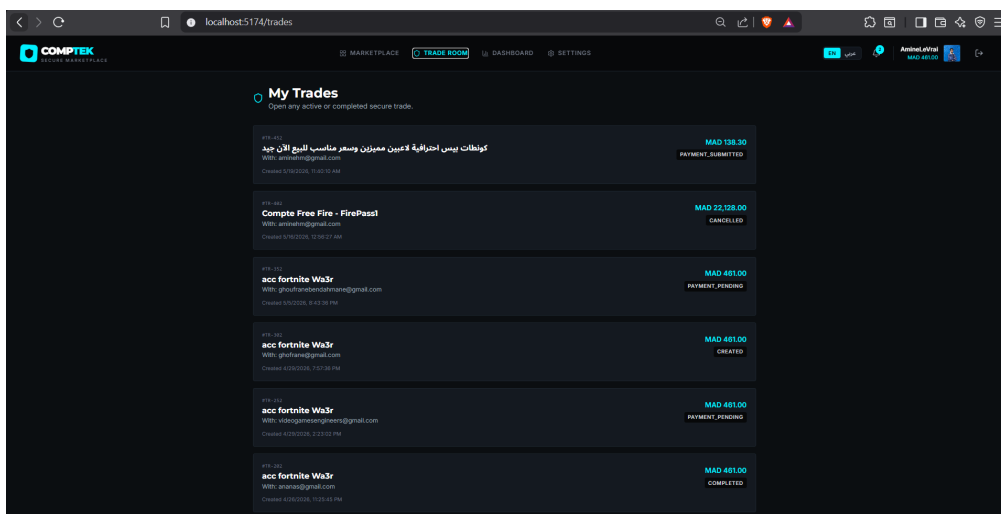


Figure 3.9: My trades (/trades).

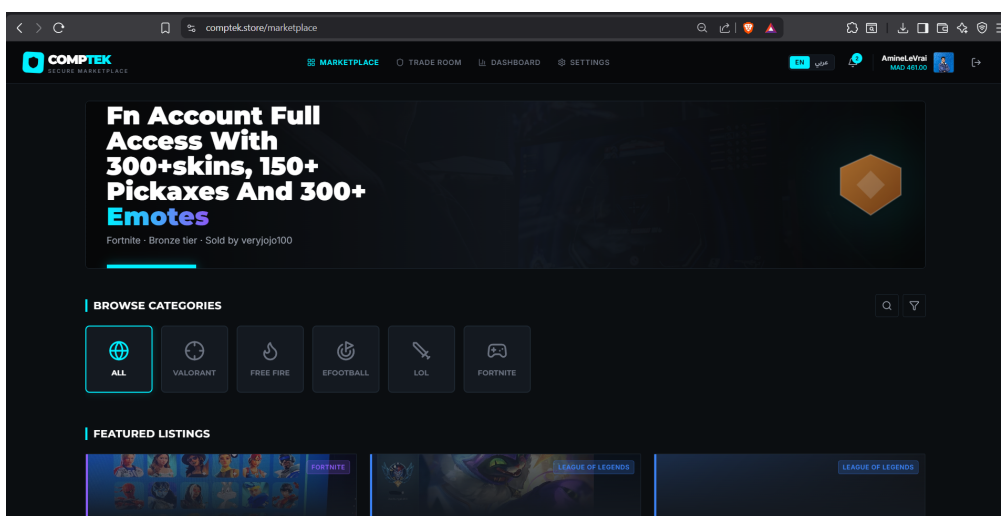


Figure 3.10: Seller dashboard (/dashboard).

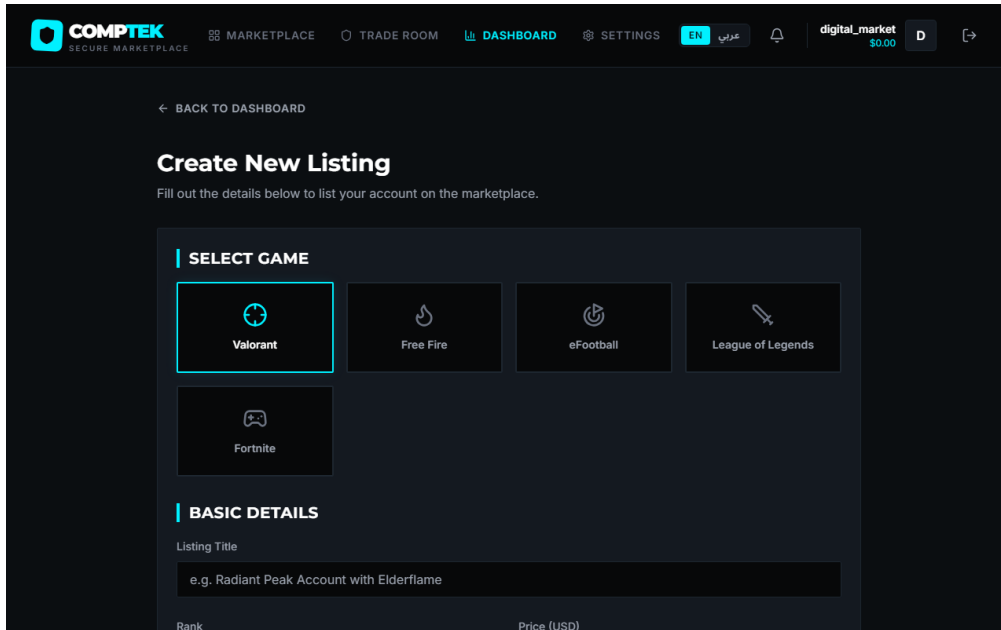


Figure 3.11: Create listing (`/sell`) with game metadata and media upload.

3.8 Persistence and Migrations

Connection pooling uses HikariCP. Notable Flyway migrations cover trade timestamps, encrypted credential columns, admin audit tables, password reset tokens, and profile currency preferences.

Supabase may additionally provide auth and row-level security concepts on the client side; the Spring API remains authoritative for trade rules in the deployed architecture described in `architecture-globale.md`.

3.9 Integration Points for Big Data

`MarketplaceEventPublisher` serializes typed events to Kafka asynchronously. Event types include listing lifecycle, failed login, rate limiting, and payment failures—see `enum MarketplaceEventType` in the backend.

`SellerRiskService` calls the risk processor REST API (`/internal/risk/seller/{id}`) with caching and optional `RISK_ALERT` notifications for medium/high bands.

3.10 Chapter Conclusion

This chapter showed how CompTek combines a **layered Spring Boot backend** with a **React presentation layer** around a strict trade workflow. The deployment view (Figure 3.1) separates OLTP (PostgreSQL) from optional stream analytics (Kafka and the Python risk processor). Three JPA class diagrams document identity, catalog, and trade aggregates; the UI screenshots illustrate how those concepts surface to buyers and sellers. Flyway-validated schemas and transactional services keep business rules server-side; the SPA focuses on clarity, real-time chat, and seller tooling. Together, these elements form a modular monolith ready for production hardening (TLS, monitoring) and for the security and Big Data topics developed in the following chapters.

Chapter 4

Application Security

Security is critical for a marketplace handling payments and account credentials. CompTek applies defense in depth: transport-level TLS in deployment, application-level authZ/authN, and field-level encryption for sensitive delivery data.

4.1 Threat Model (Simplified)

Main threats considered:

- Unauthorized access to trades, messages, or admin functions.
- Stolen or forged session tokens.
- Credential leakage from database backups or logs.
- Privilege escalation (user acting as admin or as another party in a trade).
- Abuse of public endpoints (brute force login, scraping).

4.2 Authentication

4.2.1 Email and password

Registration and login are exposed by `AuthController` (`/auth/register`, `/auth/login`) and implemented in `AuthService`. Passwords are hashed with **BCrypt** via the `PasswordEncoder` bean declared in `SecurityConfig` and injected into `AuthService`.

BCrypt and authentication provider (`SecurityConfig.java`):

Listing 4.1: BCrypt encoder and DaoAuthenticationProvider

```
1 @Bean
2 public PasswordEncoder passwordEncoder() {
3     return new BCryptPasswordEncoder();
4 }
5
6 @Bean
7 public DaoAuthenticationProvider authenticationProvider() {
8     DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
9     authProvider.setUserDetailsService(customUserService);
10    authProvider.setPasswordEncoder(passwordEncoder());
11    return authProvider;
```

12 }

Registration (AuthService.java): encode password, assign BUYER role, create Profile, return JWT:

Listing 4.2: Register flow

```
1 user.setPassword(passwordEncoder.encode(request.password()));
2 user.setRoles(Set.of(defaultRole));
3 userRepository.save(user);
4 String token = jwtService.generateToken(
5     customUserService.loadUserByUsername(user.getEmail()));
6 return AuthResponse.bearer(token, TOKEN_TTL_MS);
```

Login (AuthController + AuthService): rate limiting on IP/email, then AuthenticationManager validates credentials; failed logins emit LOGIN_FAILED events:

Listing 4.3: Login endpoint and response DTO

```
1 // AuthController.java
2 authRateLimiterService.assertAllowed(extractClientIp(servletRequest), request.
   email());
3 return ResponseEntity.ok(authService.login(request));
4
5 // AuthResponse.java
6 public record AuthResponse(String token, String tokenType, long expiresInMs) {
7     public static AuthResponse bearer(String token, long expiresInMs) {
8         return new AuthResponse(token, "Bearer", expiresInMs);
9     }
10 }
```

Password reset: /auth/forgot-password and /auth/reset-password are public; PasswordResetService stores a hashed token, enforces TTL, and emails a reset link to the frontend.

4.2.2 JWT session model

The API is **stateless** (SessionCreationPolicy.STATELESS). JwtService signs tokens (HS256, subject = email, TTL from security.jwt.expiration-ms):

Listing 4.4: JWT generation (JwtService.java)

```
1 return Jwts.builder()
2     .subject(userDetails.getUsername())
3     .issuedAt(now)
4     .expiration(expiration)
5     .signWith(getSignInKey(), Jwts.SIG.HS256)
6     .compact();
```

JwtAuthenticationFilter reads Authorization: Bearer ..., validates the token, and sets SecurityContextHolder:

Listing 4.5: JWT filter (JwtAuthenticationFilter.java)

```
1 String header = request.getHeader("Authorization");
2 if (header == null || !header.startsWith("Bearer ")) {
3     chain.doFilter(request, response);
4     return;
5 }
6 String token = header.substring(7);
7 String email = jwtService.extractUsername(token);
```

```

8 UserDetails userDetails = userService.loadUserByUsername(email);
9 if (jwtService.isTokenValid(token, userDetails)) {
10     SecurityContextHolder.getContext().setAuthentication(
11         new UsernamePasswordAuthenticationToken(
12             userDetails, null, userDetails.getAuthorities());
13     }
14 chain.doFilter(request, response);

```

4.2.3 Supabase bridge

POST /auth/supabase accepts a Supabase access token; AuthService.loginWithSupabase validates it, creates or loads User, syncs Profile, then returns a CompTek JWT:

Listing 4.6: Supabase bridge (AuthService.java)

```

1 SupabaseUserInfo supabaseUser = supabaseAuthService.fetchUser(accessToken);
2 User user = userRepository.findByEmail(supabaseUser.email())
3     .orElseGet(() -> createFromSupabase(supabaseUser));
4 syncProfileFromSupabase(user, supabaseUser);
5 return AuthResponse.bearer(
6     jwtService.generateToken(
7         customUserService.loadUserByUsername(user.getEmail()),
8         TOKEN_TTL_MS);

```

4.3 Authorization (RBAC)

SecurityConfig.securityFilterChain defines HTTP access (excerpt):

Listing 4.7: Security filter chain (SecurityConfig.java)

```

1 .sessionManagement(s -> s.sessionCreationPolicy(SessionCreationPolicy.
   STATELESS))
2 .authorizeHttpRequests(auth -> auth
3     .requestMatchers("/auth/login", "/auth/register", "/auth/supabase",
4         "/auth/forgot-password", "/auth/reset-password", "/ws/**").permitAll()
5     .requestMatchers(HttpMethod.GET, "/listings/**", "/categories/**",
6         "/payment-methods/**", "/reviews/seller/**", "/currency/**").permitAll
   ()
7     .requestMatchers("/admin/**").hasRole("ADMIN")
8     .requestMatchers("/dashboard/**", "/seller/**").hasRole("SELLER")
9     .anyRequest().authenticated())
10 .addFilterBefore(new JwtAuthenticationFilter(jwtService, customUserService),
11     UsernamePasswordAuthenticationFilter.class)

```

Level	Routes
Public	/auth/*, /ws/**, GET /listings/**, categories, reviews
Authenticated	Trades, messages, profile, owned listings
SELLER	/dashboard/**, /seller/**
ADMIN	/admin/**

Table 4.1: RBAC aligned with SecurityConfig.

Trade role checks (buyer/seller/admin) are also enforced in TradeService.

4.4 WebSocket Security

WebSocketConfig: STOMP endpoint /ws, broker /topic, app prefix /app. HTTP allows /ws/**; JWT is required on STOMP CONNECT:

Listing 4.8: Broker and endpoint (WebSocketConfig.java)

```
1 registry.addEndpoint("/ws").setAllowedOriginPatterns(  
2     "http://localhost:5173", "https://www.comptek.store",  
3     "https://comptek.vercel.app", "https://react-comptek-code-*.vercel.app");  
4 registry.enableSimpleBroker("/topic");  
5 registry.setApplicationDestinationPrefixes("/app");
```

Listing 4.9: JWT on STOMP CONNECT

```
1 if (StompCommand.CONNECT.equals(accessor.getCommand())) {  
2     String token = resolveBearerToken(accessor);  
3     String email = jwtService.extractUsername(token);  
4     UserDetails userDetails = customUserService.loadUserByUsername(email);  
5     if (!jwtService.isTokenValid(token, userDetails)) {  
6         throw new BadCredentialsException("Invalid websocket token");  
7     }  
8     accessor.setUser(new StompPrincipal(email));  
9 }
```

Clients subscribe to /topic/trade/{tradeId}; TradeWebSocketController restricts messaging to trade participants.

4.5 Protection of Delivered Credentials

When the seller delivers game account credentials, plaintext never goes to the database. TradeService.deliverCredentials builds a JSON payload, encrypts it with **AES-256-GCM**, and stores only the ciphertext in trade_credentials.encrypted_payload.

AES-256-GCM encrypts trade credentials with a 256-bit secret key and GCM mode, which both hides the data and detects modification of the stored ciphertext before decryption.

Listing 4.10: AES-256-GCM setup and encrypt

```

1 private static final String ALGO = "AES/GCM/NoPadding";
2 private static final int IV_LEN = 12;
3 private static final int TAG_BITS = 128;
4
5 public TradeCredentialCryptoService(
6     @Value("${security.trade.credentials.key:...}") String base64Key) {
7     byte[] rawKey = Base64.getDecoder().decode(base64Key);
8     if (rawKey.length != 32) {
9         throw new IllegalStateException(
10             "security.trade.credentials.key must be 32 bytes base64 (AES-256)"
11         );
12     }
13     this.key = new SecretKeySpec(rawKey, "AES");
14 }
15
16 public String encrypt(String plaintext) {
17     byte[] iv = new byte[IV_LEN];
18     secureRandom.nextBytes(iv);
19     Cipher cipher = Cipher.getInstance(ALGO);
20     cipher.init(Cipher.ENCRYPT_MODE, key, new GCMParameterSpec(TAG_BITS, iv));
21     byte[] ciphertext = cipher.doFinal(plaintext.getBytes(StandardCharsets.
22         UTF_8));
23     return Base64.getEncoder().encodeToString(iv) + ":"
24         + Base64.getEncoder().encodeToString(ciphertext);
25 }

```

Persistence (entity/TradeCredential.java): one row per trade, encrypted_payload column only:

Listing 4.11: TradeCredential entity

```

1 @OneToOne
2 @JoinColumn(name = "trade_id", nullable = false, unique = true)
3 private Trade trade;
4
5 @Column(name = "encrypted_payload", nullable = false, length = 8000)
6 private String encryptedPayload;

```

Usage in trade flow (TradeService.java): seller-only in DELIVERY_PENDING; buyer decrypts in revealCredentials:

Listing 4.12: Encrypt on delivery, decrypt on reveal

```

1 String encrypted = tradeCredentialCryptoService.encrypt(toJson(payload));
2 credential.setEncryptedPayload(encrypted);
3 tradeCredentialRepository.save(credential);
4
5 // Buyer reveal:
6 CredentialPayload payload = fromJson(
7     tradeCredentialCryptoService.decrypt(credential.getEncryptedPayload()));

```

Operators must inject the Base64 key via environment / .env.properties, never commit production keys. Rotation would require re-encrypting existing rows (future work).

4.6 Data Validation and Error Handling

Controllers validate input before services run. Example: TradeController requires a valid body for trade creation:

Listing 4.13: @Valid on controller (TradeController.java)

```
1 @PostMapping
2 public TradeResponse create(@Valid @RequestBody TradeCreateRequest request,
3                             Authentication authentication) { ... }
4
5 // TradeCreateRequest.java
6 public record TradeCreateRequest(
7     @NotNull Long listingId,
8     @NotNull Long paymentMethodId
9 ) {}
```

MarketplaceEvent (Kafka contract) uses @NotBlank / @NotNull on required JSON fields

Global handler (exception/GlobalExceptionHandler.java) maps exceptions to JSON without stack traces—only timestamp, status, error, message:

Listing 4.14: Validation and error envelope

```
1 @ExceptionHandler(MethodArgumentNotValidException.class)
2 public ResponseEntity<?> handleValidation(MethodArgumentNotValidException ex)
3 {
4     String msg = ex.getBindingResult().getFieldErrors().stream()
5         .map(fe -> fe.getField() + ": " + fe.getDefaultMessage())
6         .findFirst().orElse("Validation failed");
7     return error(HttpStatus.BAD_REQUEST, msg);
8 }
9 private ResponseEntity<Map<String, Object>> error(HttpStatus status, String
10 message) {
11     return ResponseEntity.status(status.value()).body(Map.of(
12         "timestamp", Instant.now().toString(),
13         "status", status.value(),
14         "error", status.getReasonPhrase(),
15         "message", message));
16 }
```

Also handled: BadCredentialsException (401), locked/disabled accounts (403), SecurityException (forbidden trade actions), IllegalStateException (409).

4.7 Kafka and Internal APIs

Where in the codebase: service/MarketplaceEventPublisher.java (Kafka producer), dto/MarketplaceEvent + dto/MarketplaceEventType.java (event contract), service/AuthService.java, ListingService.java (publishers), service/SellerRiskService.java (HTTP client to risk processor), config/RiskBridgeProperties (app.risk-bridge.*), config/KafkaProducerConfig.java (optional producer bean).

Non-blocking Kafka publish. Feature flag app.kafka.producer.enabled; failures log a warning and do not roll back the user transaction:

Listing 4.15: MarketplaceEventPublisher.java

```
1 public void publish(String key, MarketplaceEvent event) {
2     if (!enabled) {
3         return;
4     }
5     kafkaTemplate.send(topic, key, event).whenComplete((result, throwable) ->
6     {
7         if (throwable != null) {
8             log.warn("Failed to publish event type={} key={}: {}",
9                 event.getType(), key, throwable.getMessage());
10         }
11     });
12 }
```

```

8         event.eventType(), key, throwable.getMessage());
9     }
10    });
11 }

```

Events carry typed metadata (e.g. LOGIN_FAILED from AuthService) with IP/device from RequestContextSer schema fields are validated on the DTO.

Internal risk API (not public). SellerRiskService calls the Python risk-processor on the Docker network when app.risk-bridge.enabled=true:

Listing 4.16: Risk bridge HTTP call (SellerRiskService.java)

```

1 String uri = base + "/internal/risk/seller/" + user.getId();
2 HttpRequest request = HttpRequest.newBuilder()
3     .uri(URI.create(uri))
4     .timeout(Duration.ofMillis(bridge.getTimeoutMs()))
5     .GET()
6     .build();
7 HttpResponse<String> response = httpClient.send(request,
8     HttpResponse.BodyHandlers.ofString());

```

This endpoint is not exposed in SecurityConfig as a public route; only the Spring backend calls it server-to-server. Consumers (Kafka, HBase, HDFS) run in Comptek-BigData-microservice-1 on an internal network.

4.8 Security Configuration Summary

Table 4.2 summarizes mechanisms.

Concern	Mechanism in CompTek
Password storage	BCrypt hashing
API authentication	JWT (stateless), JwtAuthenticationFilter
OAuth / social	Supabase token validation → CompTek JWT
Authorization	Spring Security URL rules + trade-level checks
Chat	STOMP + JWT at connect
Credentials at rest	AES-256-GCM (TradeCredentialCryptoService)
Admin actions	ADMIN role + AdminAuditEvent trail
Fraud signals	Marketplace events + risk score bands

Table 4.2: Security mechanisms deployed in CompTek.

Chapter 5

Big Data Pipeline

This chapter explains the optional analytics and fraud-scoring pipeline: how events leave the OLTP system, how they are processed in near real time, and how results flow back to the marketplace.

5.1 Motivation

Operational databases excel at transactional consistency but are a poor fit for high-volume historical analytics and cross-user pattern detection. CompTek therefore adopts an **event-driven** design: significant actions become immutable events on Kafka, consumed by a dedicated **risk processor** while the user request path stays fast.

Goals of the pipeline:

- Compute a **seller risk score** from recent behavior (listing velocity, failed payments, etc.).
- Persist **raw history** on HDFS for replay and batch analysis.
- Expose **low-latency lookups** via HBase for the Java API.
- Support **Hive** queries for reporting and trend dashboards (admin / analytics).

5.2 Architecture

Figure 5.1 shows the end-to-end Big Data pipeline. The `Comptek-BigData-microservice-1` repository provides `docker-compose.yml` with services including:

- **Kafka** (KRaft, no Zookeeper) — topics `marketplace-events`, `risk-events`, `risk-dlq`.
- **HDFS** (namenode, datanode) — data lake paths for raw/processed data.
- **Hive** (metastore, server) — SQL analytics on HDFS datasets.
- **HBase** — latest risk score per seller entity.
- **risk-processor** — Python container (Flask API + Kafka consumer loop).

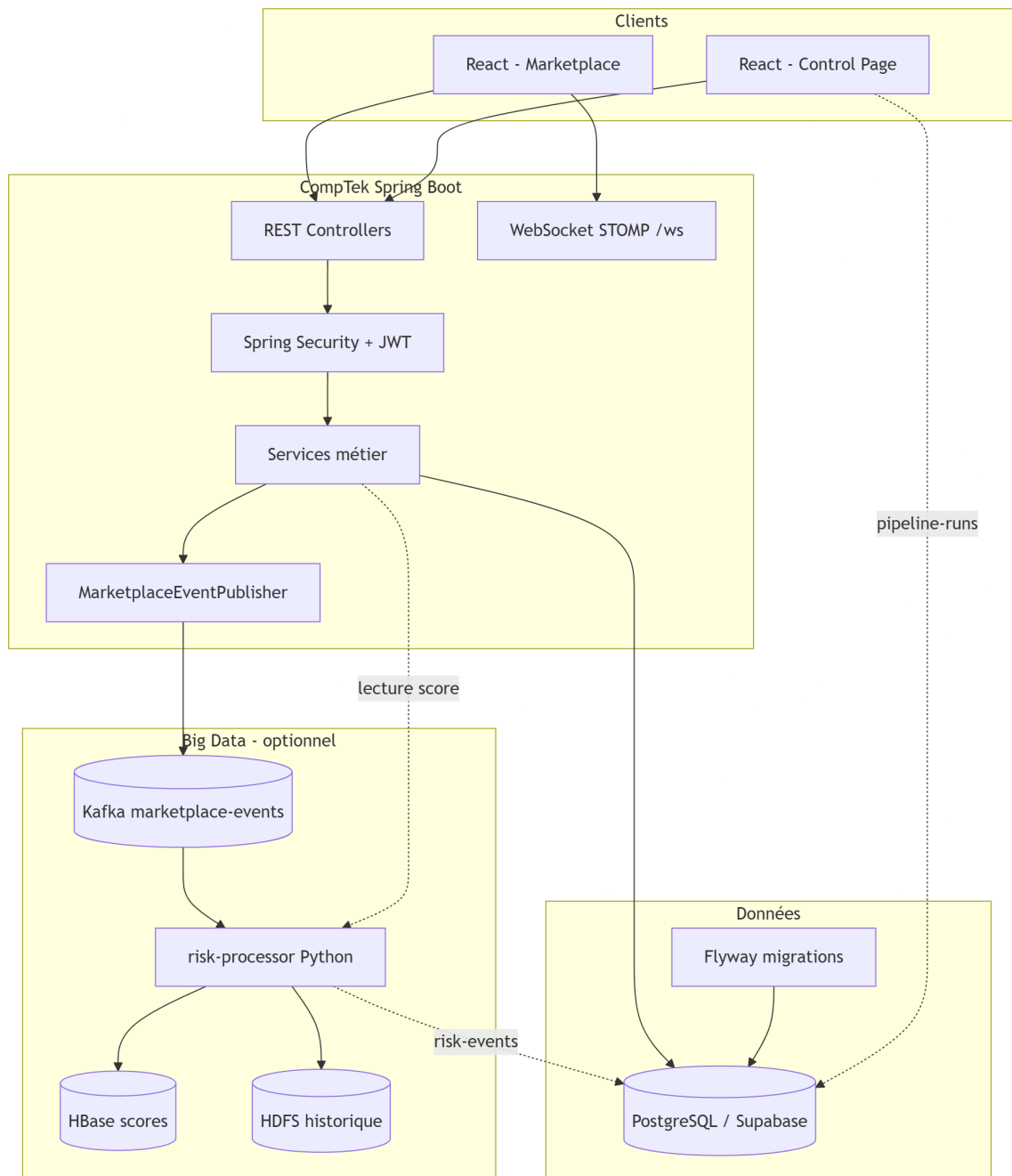


Figure 5.1: Big Data pipeline architecture.

End-to-end flow:

1. Spring MarketplaceEventPublisher sends JSON events to marketplace-events.
2. risk-processor consumes, parses, and applies update_and_score (rule engine).
3. Scores are written to HBase; derived events may be published to risk-events.
4. Failed or malformed messages go to risk-dlq for inspection.
5. SellerRiskService in Spring queries Flask at /internal/risk/seller/{id} when RISK_BRIDGE_ENABLED is true.

5.3 Event Contract

All events share a canonical schema (JSON Schema in the backend docs). Important fields:

- `event_id`, `event_type`, `event_time`
- `user_id` (seller or actor), optional `listing_id`
- `ip_address`, `device_id` when available
- `metadata` for extensibility

Representative `event_type` values: `listing_created`, `listing_updated`, `login_failed`, `order_payment_failed`, rate-limit signals, and review-related types planned in the roadmap.

Validation on the producer (Jakarta) reduces poison messages; the consumer still guards with a dead-letter topic.

5.4 Rule-Based Risk Scoring (MVP)

The MVP deliberately avoids machine learning to keep the system explainable for academic evaluation. Implementation lives in `Comptek-BigData-microservice-1/risk-processor/app/`:

- `scoring.py` — `update_and_score(event)` computes the fraud score from in-memory state.
- `main.py` — Kafka consumer on `marketplace-events`; calls `scoring`, writes HBase, publishes `risk-events`.
- `risk_api.py` — Flask endpoint that reads the latest score from HBase for Spring Boot.
- `hbase_client.py` — `upsert_risk` and `get_seller_risk` persistence helpers.

5.4.1 Score calculation (`scoring.py`)

Each `user_id` keeps a **30-minute sliding window** (`WINDOW_MINUTES = 30`): dequeues of listing and failed-payment timestamps, plus cumulative sets of IP addresses and device IDs. On every event, counters are updated, old entries are trimmed, then points are added when thresholds are met.

Rule	Threshold	Points
Many listings in 30 min	≥ 5 <code>listing_created</code>	+30
Many failed payments	≥ 5 <code>order_payment_failed</code>	+30
Many distinct IPs	≥ 5 IPs seen	+20
Many distinct devices	≥ 5 devices seen	+20

Table 5.1: Fraud score rules (maximum theoretical score = 100).

Bands: low if `score < 40`; medium if `40 ≤ score < 70`; high if `score ≥ 70`. The function returns `entity_id`, `score`, `band`, and a `reasons` list (e.g. `many_listings_short_time`).

Listing 5.1: Core scoring logic (`scoring.py`)

```
1 WINDOW_MINUTES = 30
2
3 def update_and_score(event):
4     now = _parse_event_time(event["event_time"])
5     user_id = event["user_id"]
6     s = state[user_id]
7
```

```

8     if event["event_type"] == "listing_created":
9         s["listing_times"].append(now)
10    if event["event_type"] == "order_payment_failed":
11        s["failed_payments"].append(now)
12
13    _trim(s["listing_times"], now)
14    _trim(s["failed_payments"], now)
15
16    if event.get("ip_address"):
17        s["ips"].add(event["ip_address"])
18    if event.get("device_id"):
19        s["devices"].add(event["device_id"])
20
21    score = 0
22    reasons = []
23    if len(s["listing_times"]) >= 5:
24        score += 30
25        reasons.append("many_listings_short_time")
26    if len(s["failed_payments"]) >= 5:
27        score += 30
28        reasons.append("many_failed_payments")
29    if len(s["ips"]) >= 5:
30        score += 20
31        reasons.append("many_ips")
32    if len(s["devices"]) >= 5:
33        score += 20
34        reasons.append("many_devices")
35
36    band = "low" if score < 40 else "medium" if score < 70 else "high"
37    return {
38        "entity_id": user_id,
39        "entity_type": "seller",
40        "score": score,
41        "band": band,
42        "reasons": reasons,
43        "event_time": event["event_time"],
44    }

```

After scoring, `main.py` persists the result to HBase (`upsert_risk`) and may publish JSON to topic `risk-events`.

5.4.2 Exposing the score to CompTek (`risk_api.py`)

`risk_api.py` does **not** recalculate the score; it serves the last value stored in HBase table `risk_scores` (row key `seller#<user_id>`). Spring `SellerRiskService` calls this internal endpoint when `RISK_BRIDGE_ENABLED` is true.

Listing 5.2: Risk read API (`risk_api.py`)

```

1 @app.get("/internal/risk/seller/<user_id>")
2 def risk_seller(user_id):
3     cleaned = (user_id or "").strip()
4     if not cleaned:
5         return jsonify({"found": False, "message": "user_id required"}), 400
6     row = get_seller_risk(cleaned)
7     if row is None:
8         return jsonify({"found": False, "userId": cleaned}), 404
9     return jsonify(row), 200

```

`get_seller_risk` returns `found`, `userId`, `score`, `band`, `reasons`, and `updatedAt` fields mapped in Java to `RiskSnapshotResponse` for the seller UI.

5.5 Storage Layers

CompTek follows a classic **lambda-style split**: Kafka carries events in near real time; **HBase** serves the operational read path for the Java API; **HDFS** and **Hive** are the batch/analytics tier for history and SQL reporting. PostgreSQL (Supabase) remains the system of record for marketplace transactions—the Hadoop stack is optional and does not replace OLTP.

After each successful score, `main.py` calls `upsert_risk` in `hbase_client.py`. The processor connects to container `hbase` via HappyBase (Thrift) and writes to table `risk_scores`.

- **Row key (seller risk):** `seller#<user_id>`
- **Column family:** `risk` — `score`, `band`, `reasons` (JSON), `updated_at`
- **Row key (audit):** `login_failed#<event_id>` when `event_type == login_failed` (store `login_failed`)

Spring Boot does not talk to HBase directly. `SellerRiskService` calls `Flask /internal/risk/seller/<id>`, which reads the same row via `get_seller_risk`. Retries on put/get reconnect the Thrift client if HBase is temporarily unavailable.

Listing 5.3: HBase upsert for latest seller risk (`hbase_client.py`)

```
1 def upsert_risk(result):
2     row_key = f"seller#{result['entity_id']}".encode("utf-8")
3     _put_with_retry(
4         row_key,
5         {
6             b"risk:score": str(result["score"]).encode("utf-8"),
7             b"risk:band": result["band"].encode("utf-8"),
8             b"risk:reasons": json.dumps(result["reasons"]).encode("utf-8"),
9             b"risk:updated_at": datetime.now(timezone.utc).isoformat().encode(
10                 "utf-8"),
11         },
12     )
```

This design favors **low-latency, key-based lookup** (one seller, one current score) over relational joins—appropriate for fraud flags on the seller profile UI.

5.5.1 HDFS — data lake (infrastructure ready, sink planned)

`docker-compose.yml` starts a Hadoop 3 cluster: `namenode` (`hdfs://namenode:8020`, UI port 9870) and `datanode`, with Docker volumes `namenode_data` and `datanode_data` for persistence across restarts.

The intended layout (documented in `docker-kafka-fraud-risk-implementation-guide.md`) partitions raw marketplace JSON by date and hour, for example: `/data/raw/events/dt=YYYY-MM-DD/hr=HH/event_id`. That supports replay if the consumer is redeployed, seasonal fraud analysis, and training future models.

Current code status: module `hdfs_sink.py` exists but is not implemented yet; the consumer loop in `main.py` does not call an HDFS writer. HDFS is therefore part of the **target architecture** and local demo stack, not of the live write path in the MVP commit.

5.5.2 Hive — SQL analytics on HDFS (infrastructure ready)

Hive is wired in Compose: `hive-metastore`, `hive-server` (HiveServer2 port 10000), and a dedicated PostgreSQL metastore (`hive-metastore-postgresql`). Tables would point at HDFS paths (e.g. external table over `/data/raw/events`) so admins could run queries such as daily listing velocity or failed-payment counts by seller.

No Hive DDL or load jobs are checked into `Comptek-BigData-microservice-1` yet; once `hdfs_sink.py` writes partitioned files, Hive becomes the reporting layer on top of the lake.

5.5.3 Summary

Store	Purpose	MVP in repo
Kafka	Stream ingest, DLQ	Consumer + producer in <code>main.py</code>
HBase	Latest risk + login-failed audit rows	<code>hbase_client.py</code>
HDFS	Immutable event history	Compose only; <code>hdfs_sink.py</code> stub
Hive	Batch SQL / dashboards	Compose only

Table 5.2: Storage roles in the Big Data stack.

Chapter 6

General Conclusion

This report documented *CompTek*, a Moroccan-oriented marketplace for video game accounts developed in the framework of the Master Big Data & Internet of Things at ENSAM. The project was conceived as more than a catalog application: it is a response to the trust gap that characterizes informal peer-to-peer trading—payment disputes, opaque credential handover, weak traceability, and fraud patterns that only become visible when behavior is observed at scale. By combining a structured trade workflow, application security, and an optional stream-oriented analytics stack, CompTek demonstrates how enterprise Java engineering and modern data processing can reinforce one another in a single product.

6.1 Synthesis of the Work

The four technical chapters of this report map directly onto the competencies targeted by the curriculum and onto the repositories that implement the platform.

Fundamental concepts anchored the vocabulary of the domain: listings, bilateral trades with an explicit state machine, marketplace events, and the lambda-style split between OLTP (PostgreSQL) and analytics (Kafka, HBase, HDFS, Hive). These notions are not abstract; they are enforced in code—for example through `TradeService` transition rules, `TradeEventLog` auditing, and the canonical `MarketplaceEvent` contract published to Kafka.

The JEE application delivered the operational core of CompTek. Spring Boot 3.3 on Java 21 provides a layered backend (controllers, DTOs, services, repositories, entities) with Flyway-managed schema evolution and Hibernate validation. The React client offers marketplace discovery, listing management, trade rooms with STOMP chat, and a seller dashboard. Together they form a modular monolith that keeps business rules server-side while presenting a clear, real-time experience to buyers and sellers. The deployment view and JPA class diagrams, supported by UI screenshots, show that the implementation matches the intended three-tier architecture.

Application security addressed the sensitivity of the use case: account credentials, payment-related steps, and administrative moderation. Defense in depth was applied through BCrypt password hashing, stateless JWT authentication (including a Supabase OAuth bridge), role-based access control, AES-256-GCM encryption for delivered credentials, secured WebSocket connections, and an admin audit trail. The threat model considered unauthorized access, token abuse, credential leakage, privilege escalation, and endpoint abuse—each mitigated by concrete mechanisms summarized in Table 4.2.

The Big Data pipeline extended the marketplace without blocking transactional traffic. When enabled, `MarketplaceEventPublisher` emits typed events to Kafka; the Python risk processor maintains sliding-window counters per seller, applies interpretable scoring rules, persists results to HBase, and optionally publishes to `risk-events`. Spring `SellerRiskService` reads the latest score through an internal REST bridge, surfacing risk bands to the seller UI. Feature flags (`KAFKA_PRODUCER_ENABLED`, `RISK_BRIDGE_ENABLED`) preserve developer ergonomics: the marketplace can run locally without the full Hadoop stack, while Docker Compose orchestrates the complete analytics environment for demonstrations and further work.

6.2 Objectives Achieved and Remaining Gaps

Relative to the objectives stated in the general introduction, the delivered V1 product meets the functional goals of listing publication, guided trade flow, real-time communication, encrypted credential delivery, authentication and profiles, admin moderation, and rule-based seller risk scoring. On the technical side, the project achieves a clear layered JEE design, Flyway migrations, a canonical Kafka event contract, Docker-based deployment of the analytics stack, and HBase-backed low-latency risk lookup—with HDFS and Hive provisioned as the batch tier.

Several items were deliberately scoped out or left as infrastructure-ready stubs, as documented throughout the report. There is no licensed financial escrow in the current version; dispute resolution remains procedural rather than legal. Fraud detection relies on transparent rules rather than machine-learning models, which favors explainability in an academic MVP but limits adaptability to evolving attack patterns. The HDFS sink (`hdfs_sink.py`) and Hive external tables are part of the target architecture—Compose brings up namenode, datanode, metastore, and `HiveServer2`—but the live write path in the repository stops at Kafka consumption, HBase upserts, and the risk API. Closing that gap would unlock historical replay, seasonal analysis, and SQL dashboards on top of the data lake.

6.3 Lessons Learned

Beyond feature checklists, the project yielded practical lessons for full-stack and data-oriented development.

- **Workflow-first design reduces disputes.** Modeling trades as a state machine with role-based transitions and persistent event logs is more effective than ad hoc status fields; it gives moderators and developers a shared language for what happened and what is allowed next.
- **Security and UX must share one identity model.** Reusing JWT validation for REST and STOMP avoids split-brain sessions and simplifies the React client.
- **Analytics should be asynchronous and optional.** Publishing events after the transactional commit path, with non-blocking Kafka sends and feature flags, prevents the Big Data stack from becoming a single point of failure for marketplace operations.
- **Interpretable risk scores build trust with stakeholders.** Bands (`low`, `medium`, `high`) and explicit `reasons` arrays make seller-facing alerts defensible; they also provide a baseline before investing in ML-based detectors.
- **Documentation and reproducibility matter.** Architecture guides, Docker Compose manifests, and this report together allow another team to redeploy and extend the system without reverse-engineering decisions from scattered commits alone.

6.4 Perspectives

CompTek is positioned as a solid foundation rather than a finished commercial product. Short-term improvements include implementing the HDFS event sink and Hive DDL for operational reporting, hardening production deployment (TLS termination, secrets management, centralized logging and metrics), and expanding automated test coverage across trade transitions and risk scoring edge cases.

Medium-term evolution could introduce integration with a regulated payment provider for true escrow, richer moderation workflows (appeals, timed holds), and multi-game catalog features driven by admin-configured metadata. On the analytics side, migrating from rule-based scores to supervised models trained on labeled fraud cases—once sufficient history exists on HDFS—would complement the existing interpretable layer rather than replace it abruptly.

From a research and teaching perspective, the project successfully illustrates the Master program’s intersection of **JEE**, **security**, and **Big Data**: students build a credible marketplace, protect high-value data, and wire a stream pipeline that consumes the same events users generate during normal activity. That alignment between product narrative and technical depth is, in our view, the principal outcome of this work.

6.5 Closing Remarks

Informal markets for digital goods will continue to exist; the challenge is not to eliminate them but to offer a safer structure around them. CompTek proposes such a structure for Moroccan users: prices in MAD, a visible trade lifecycle, encrypted handover, and data-driven signals when seller behavior departs from the norm. The codebase across the Spring backend, React client, and Big Data microservice is open to iteration; the architecture described here is intended to guide that iteration with clarity.

We thank again our supervising professors and the ENSAM faculty for their support throughout this academic year. We hope this report serves both as a record of what was built and as a reference for those who will extend CompTek in future semesters.