# Elaris Marketplace — Tài liệu hệ thống toàn diện

*Pixel Pet Trading System* — tài liệu này chứa **ý tưởng, kiến trúc, thiết kế chi tiết, API contract, data model, event contract, flow saga, deploy, observability, bảo mật, roadmap, testing**

Contents

[Elaris Marketplace — Tài liệu hệ thống toàn diện 1](#_Toc212020607)

[Tổng quan ý tưởng (elevator pitch) 1](#_Toc212020608)

[Tính năng chính (MVP → mở rộng) 1](#_Toc212020609)

[MVP 1](#_Toc212020610)

[Tính năng mở rộng 1](#_Toc212020611)

[Kiến trúc tổng thể (text + ASCII diagram) 2](#_Toc212020612)

[Dịch vụ & trách nhiệm (chi tiết) 2](#_Toc212020613)

[Gateway (YARP + .NET 8) 2](#_Toc212020614)

[YARP features to enable 2](#_Toc212020615)

[Identity (Duende IdentityServer) 2](#_Toc212020616)

[ProductService (MongoDB) 3](#_Toc212020617)

[Basket (Redis) 3](#_Toc212020618)

[Inventory (Postgres / EF Core) 3](#_Toc212020619)

[Ordering (CQRS + EF Core + Read model) 3](#_Toc212020620)

[Payment 3](#_Toc212020621)

[Orchestrator (Saga) 4](#_Toc212020622)

[Cross-cutting 4](#_Toc212020623)

[Data models (sketch) 4](#_Toc212020624)

[Product (Mongo document) 4](#_Toc212020625)

[Inventory (Postgres) 4](#_Toc212020626)

[Basket (Redis) 5](#_Toc212020627)

[Event & Message Contracts (JSON examples) 5](#_Toc212020628)

[gRPC API (Inventory check) — sample proto 7](#_Toc212020629)

[Ordering Saga (MassTransit) — state machine sketch (C#) 8](#_Toc212020630)

[Outbox & Inbox (Exactly-once semantics) 8](#_Toc212020631)

[Auth & Security details 9](#_Toc212020632)

[Idempotency & Concurrency 9](#_Toc212020633)

[Observability & Monitoring 9](#_Toc212020634)

[Testing strategy 9](#_Toc212020635)

[Deployment & Infra (recommended) 10](#_Toc212020636)

[API Design (samples) 10](#_Toc212020637)

[ProductService (Minimal API examples) 10](#_Toc212020638)

[Basket 10](#_Toc212020639)

[Ordering 10](#_Toc212020640)

[Payment 11](#_Toc212020641)

[Code snippets — Quickstart hints 11](#_Toc212020642)

[Minimal API endpoint (Product) 11](#_Toc212020643)

[MassTransit consumer registration 11](#_Toc212020644)

[Dev roadmap & milestones 11](#_Toc212020645)

[Phase 0 — Setup (1 week) 11](#_Toc212020646)

[Phase 1 — Core services MVP (3–4 weeks) 11](#_Toc212020647)

[Phase 2 — Inventory & Ordering (3–4 weeks) 12](#_Toc212020648)

[Phase 3 — Saga & Payment (2–3 weeks) 12](#_Toc212020649)

[Phase 4 — Observability & Reliability (1–2 weeks) 12](#_Toc212020650)

[Phase 5 — Polish & Docs (1–2 weeks) 12](#_Toc212020651)

[Acceptance criteria (how to demo) 12](#_Toc212020652)

[Rủi ro & Mitigation 12](#_Toc212020653)

[Dev tips & libs 13](#_Toc212020654)

[UI / UX suggestions (pixel vibe) 13](#_Toc212020655)

## Tổng quan ý tưởng (elevator pitch)

**Elaris Marketplace** là một chợ ảo nơi người dùng mua/bán/trade “pet” và vật phẩm pixel. Hệ thống mô phỏng luồng mua hàng thực tế để bạn thực hành toàn bộ patterns microservice: gateway, auth (Duende), product (Mongo), basket (Redis), inventory (Postgres), ordering (CQRS + Outbox + Saga + MassTransit), payment (tách service), orchestrator (Saga), observability (OpenTelemetry), resilience (Polly), caching/rate-limiting(YARP).  
Mục tiêu: học kiến trúc, implement patterns production-ready, có UI pixel-art đỉnh.

## Tính năng chính (MVP → mở rộng)

### MVP

* Đăng ký/đăng nhập (Duende + JWT RS256 + refresh rotation)
* Quản lý sản phẩm/pet: danh sách, chi tiết, filter/sort/paging (ProductService)
* Giỏ hàng (Redis), checkout → order flow
* Inventory reservation/release (sync via gRPC + events)
* Ordering service xử lý orders, saga orchestrated → final status
* Payment service pre-authorize & capture (simulated)
* Observability: traces + metrics + centralized logs
* Rate limit & caching via API Gateway

### Tính năng mở rộng

* Marketplace: seller listing, auctions, offers
* Trade between users
* NFT/pixel art minting (tương lai)
* Leaderboard, events, promotions

## Kiến trúc tổng thể (text + ASCII diagram)

A computer screen with white text

AI-generated content may be incorrect.

* Services talk sync via HTTP/gRPC when cần low latency (e.g., check stock via gRPC).
* Async events via RabbitMQ (MassTransit).
* Outbox pattern đảm bảo reliable event publishing.
* Gateway (YARP) thực hiện authentication passthrough / BFF cookie support.

## Dịch vụ & trách nhiệm (chi tiết)

### Gateway (YARP + .NET 8)

* Reverse proxy + OpenAPI aggregation.
* Middlewares: Rate Limiting, Output Caching, Request/Response Logging (Serilog + Enrichers).
* Auth: validate JWT, forward identity, BFF cookie support.
* Config lưu trong GitOps (YARP config) hoặc dynamic config store.

### YARP features to enable

* RateLimit per IP/user + path
* OutputCaching for public GETs (product lists)
* Forward headers for tracing (OTel)

### Identity (Duende IdentityServer)

* Users, roles (buyer, seller, admin), scopes.
* JWT RS256 (RSA key pair stored in KMS/KeyVault).
* Refresh token rotation + revoke old refresh tokens.
* Password hashing: use ASP.NET Core Identity with Argon2 (recommended) or PBKDF2 fallback.
* Support cookie-based BFF session for web UI (if choose BFF approach).

### ProductService (MongoDB)

* Document-per-aggregate: Product doc contains metadata + variants + pixel art url.
* Indexes: text index on name, description, compound index on category + price, TTL for soft-deleted archiving if needed.
* API: CRUD (minimal API + vertical slice style).
* Emits product.price-updated event when price changes.

### Basket (Redis)

* Data model: Redis Hash or RedisJSON per user key basket:{userId} with TTL.
* TTL reset on activity.
* Atomic operations using Lua or Redis transactions.
* Expose endpoints to add/remove/list/checkout.

### Inventory (Postgres / EF Core)

* Strong consistency store for stock qty.
* Expose gRPC CheckAndReserve sync for fast checks.
* Reservation logic: create reservation row with expiry timestamp; background worker (Quartz) to release expired reservations.
* Publish events inventory.reserved, inventory.released.

### Ordering (CQRS + EF Core + Read model)

* Commands write to Orders DB (transactional + outbox).
* Read model: denormalized tables for queries (e.g., order list, order detail) updated via events or direct projection.
* Use MassTransit + RabbitMQ; implement Saga State Machine for order lifecycle.
* Outbox pattern for exactly-once publish; Inbox consumer for dedup.

### Payment

* Simulated payment provider.
* Pre-authorize synchronous (gRPC/HTTP). On success -> continue; else fail.
* Final capture triggered after successful reserve & checks; emits payment.captured or payment.failed.

### Orchestrator (Saga)

* MassTransit Saga State Machine handles sequence: order.created -> reserve inventory -> pre-authorize payment -> capture -> completed or rollback flows release inventory -> refund -> canceled.
* Saga persistence (e.g., using EF Core or Mongo).

### Cross-cutting

* OpenTelemetry (OTLP) for traces/metrics.
* Logs: Serilog to centralized sink (ELK/Grafana Loki).
* Prometheus metrics, Grafana dashboard.
* Polly for resilience on sync calls.

## Data models (sketch)

### Product (Mongo document)

A computer screen with text on it

AI-generated content may be incorrect.

**Indexes**

* text: { name, description }
* { category:1, price:1 }
* { isDeleted:1 } (to quickly filter soft-deleted)

### Inventory (Postgres)

Tables:

* products (sku, name, etc.) — optional reduced copy
* inventories (inventory\_id PK, sku FK, quantity int)
* reservations (reservation\_id PK, order\_id, sku, quantity, expires\_at, status)

A screenshot of a computer program

AI-generated content may be incorrect.

### Basket (Redis)

Key: basket:{userId} value (RedisJSON):

A black screen with white text

AI-generated content may be incorrect.

TTL: 7 days (or policy).

**Orders (EF Core - Write DB)**

Entities:

* Order (Id, UserId, Total, Status, CreatedAt, Version)
* OrderLine (Id, OrderId, Sku, Price, Qty)
* OutboxMessage (for outbox)

Use optimistic concurrency (rowversion/timestamp).

## Event & Message Contracts (JSON examples)

A black rectangular object with white text

AI-generated content may be incorrect.

A black screen with a black border

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

**payment.preauthorized / payment.captured / payment.failed**  
similar shape with paymentId and statuses.

## gRPC API (Inventory check) — sample proto

A screenshot of a computer program

AI-generated content may be incorrect.

## Ordering Saga (MassTransit) — state machine sketch (C#)

A screen shot of a computer program

AI-generated content may be incorrect.

Saga actions:

* On OrderCreated: call Inventory.CheckAndReserve (send command), set timeout.
* On InventoryReserved: call Payment.PreAuthorize sync; on success, call Payment.Capture or continue to final step.
* On failure: release reservation, mark order canceled.

## Outbox & Inbox (Exactly-once semantics)

* Write business data + outbox row in same DB transaction (EF Core + Outbox table).
* Background or middleware reads outbox and publishes to RabbitMQ; mark outbox as sent.
* Consumers implement inbox dedup (store messageId) to avoid double processing.

You can use:

* EFCore.Outbox library or MassTransit Outbox (integrated).
* Saga persistence also benefit from durable storage.

## Auth & Security details

* **JWT**: RS256. Store private key in secure vault (Azure Key Vault / AWS KMS). Public key used by services.
* **Refresh token rotation**: on refresh, issue new refresh token and revoke previous; track revocation list.
* **Password hashing**: Argon2id (memory-hard). Use libs like Isopoh.Cryptography.Argon2 or Konscious.Security.Cryptography.
* **Roles & Policies**: define policies for product:write, order:manage, inventory:modify, payment:process.
* **Transport security**: mTLS internal or at least mutual verification in cluster; use TLS for gRPC.
* **Secrets**: environment variables from secret manager; do NOT store keys in repo. **Rate limiting**: YARP limit per user/IP.
* **Input validation & sanitization**: all APIs.

## Idempotency & Concurrency

* Use Idempotency-Key header for POST endpoints (checkout, payment). Persist key + result.
* Use optimistic concurrency for Orders via RowVersion.
* For critical flows (inventory decrement), do DB-level checks and row-level locking where necessary.

## Observability & Monitoring

* **Tracing**: OpenTelemetry SDK for .NET + instrument HTTP/gRPC + MassTransit instrumentation. Export OTLP to Grafana Tempo or Jaeger.
* **Metrics**: Prometheus exporter; metrics like request latencies, error rates, queue lengths, reservation counts.
* **Logging**: Serilog structured logs to Grafana Loki / Elastic.
* **Dashboards**: Grafana dashboards for key flows and system health.
* **Alerts**: use Prometheus Alertmanager: consumer lag, failed payment rate, reservation failures.

## Testing strategy

* **Unit tests** for business logic.
* **Integration tests**:
  + ProductService <-> Mongo in TestContainer.
  + Inventory <-> Postgres, check reservation timeouts.
  + MassTransit in-memory transport for Saga tests.
* **Contract tests**: ensure event schemas stable.
* **E2E tests**: simulate purchase flow via test user.
* **Chaos tests**: simulate RabbitMQ down / network delays.
* **Load tests**: k6 or Locust to simulate traffic to Gateway.

## Deployment & Infra (recommended)

* Containerize each service (Docker).
* Orchestrator: Kubernetes (AKS/EKS/GKE or local K3s).
* Use Helm charts for services.
* Messaging: RabbitMQ cluster (HA).
* DBs: Managed Postgres, Mongo Atlas (or self-managed in k8s statefulsets).
* Redis: managed or cluster with persistence if needed.
* CI/CD: GitHub Actions / Azure DevOps pipelines
  + Build + test + push images
  + Deploy via Helm
  + Canary / blue-green support for safer releases

**Secrets**: use Kubernetes Secrets backed by Vault/secret manager.

## API Design (samples)

### ProductService (Minimal API examples)

* GET /api/products?query=&page=1&pageSize=20&sort=price:asc
* GET /api/products/{sku}
* POST /api/products (admin)
* PATCH /api/products/{sku} -> emit price-updated event if price changed

### Basket

* GET /api/basket (auth)
* POST /api/basket/items { sku, qty }
* POST /api/basket/checkout -> returns orderId

### Ordering

* POST /api/orders { basket } -> creates order + emits order.created (write to Orders DB + Outbox)
* GET /api/orders/{id}

### Payment

* POST /api/payments/preauthorize { orderId, amount, paymentMethod }
* POST /api/payments/capture { paymentId }

**All POST endpoints that mutate should accept Idempotency-Key.**

## Code snippets — Quickstart hints

### Minimal API endpoint (Product)

A screen shot of a computer program

AI-generated content may be incorrect.

### MassTransit consumer registration

A computer screen with white text

AI-generated content may be incorrect.

## Dev roadmap & milestones

### Phase 0 — Setup (1 week)

* Init repo monorepo (or multi-repo). Docker + local dev compose (Postgres, Mongo, Redis, RabbitMQ).
* CI baseline, code style, secrets config.

### Phase 1 — Core services MVP (3–4 weeks)

* Identity (Duende) + basic React auth
* ProductService (Mongo) + product listing UI
* Basket (Redis) + simple UI

### Phase 2 — Inventory & Ordering (3–4 weeks)

* Inventory (Postgres) + gRPC check/reserve
* Orders basic write model + outbox
* MassTransit + RabbitMQ basic events

### Phase 3 — Saga & Payment (2–3 weeks)

* Implement Saga orchestration for order flow
* Simulated Payment service (preauthorize & capture)
* Reservation timeout worker (Quartz)

### Phase 4 — Observability & Reliability (1–2 weeks)

* OpenTelemetry tracing + Prometheus + Grafana basics
* Polly resiliency patterns
* Rate-limiting + caching at YARP

### Phase 5 — Polish & Docs (1–2 weeks)

* UI pixel art polish
* Tests, load tests, final docs, demo scripts

## Acceptance criteria (how to demo)

* User can register/login and obtain JWT.
* User can browse products, add to basket, checkout.
* On checkout: reservation created in Inventory, payment preauthorized, order completed via Saga (or canceled on failure).
* Traces across services visible in Jaeger/Tempo.
* Event messages visible in RabbitMQ and outbox processed reliably.
* Idempotent checkout: retry with same Idempotency-Key yields same result without duplication.

## Rủi ro & Mitigation

* **Eventual consistency surprises**: document UX (order pending) and implement real-time update (websocket) or polling.
* **Distributed transactions complexity**: use Saga pattern to avoid 2PC.
* **Secrets leakage**: use vaults and RBAC.
* **Race conditions on inventory**: ensure DB-level checks and reservations.
* **Scaling**: separate read models for heavy read workloads.

## Dev tips & libs

* .NET: MassTransit, EFCore, MongoDB.Driver, StackExchange.Redis, Duende.IdentityServer.
* Password hashing: Isopoh.Cryptography.Argon2 (Argon2id).
* OpenTelemetry: OpenTelemetry.Exporter.OpenTelemetryProtocol, OpenTelemetry.Instrumentation.AspNetCore.
* Test containers: DotNet.Testcontainers.
* Local dev: Docker Compose including RabbitMQ, Redis, Mongo, Postgres, and a local OTel collector.

## UI / UX suggestions (pixel vibe)

* Theme: 16-bit palette, 32x32 sprites for pets.
* Product list as grid of cards with pixel-sprite, rarity badge, price tag.
* Cart modal sliding in (pixelated border).
* Order history as "album" with pet sprite thumbnails.
* Use Tailwind + React + Canvas or CSS pixel art assets.