High Level Flight Aggregator Website – Architecture & Design Document

Goal: Build a public flight‑search aggregator that queries multiple airline / OTA APIs in real time, normalises the fares and deep‑links users to the provider to complete booking.

# 1. Introduction

The system lets travellers compare airfares quickly while demonstrating modern back‑end engineering practices (Kotlin + Spring Boot), front‑end development (Next.js + TypeScript), and a comprehensive quality strategy (unit → E2E → performance tests). All code, tools, and libraries are free/open‑source.

# 2. Purpose

Build a public-facing flight-search aggregator that:

* Queries multiple airline / OTA APIs in real-time and/or via cached feeds.
* Normalised & ranks results on price, duration, stops, CO₂ emissions etc.
* Redirects or deep-links users to the partner site to complete the booking (not as the merchant)
* Showcases modern, scalable architecture, strong testing & CI/CD for portfolio.

# 3. Scope

This high-level design covers:

|  |  |
| --- | --- |
| In Scope | Out of scope |
| Flight Search (On way, return, multi city) | Direct Payment / Ticket issuing |
| Fare rules & baggage info | Hotels, cars, activities (possible for future expansion) |
| Currency / Locale support | Loyalty Programmes |
| Basic user accounts ( Save Searches ) | Complex user management |
| Operational dashboards | Revenue Management |

# 4. System Overview

* Users search flights via a React/Next.js UI.
* The Search API Gateway checks Redis cache; on miss it calls the Aggregator Service.
* The Aggregator concurrently queries Provider Adapters (one per airline/OTA API) and returns ranked results.
* Users select an option → Redirect Service appends tracking parameters and issues a 302 to the provider. common way of performing URL redirection.
* Basic user accounts allow saved searches and notification preferences.

# 5. Architectural Design

## 5.1 Architecture Style

* Micro‑services with hexagonal / ports‑and‑adapters pattern.
* Kotlin coroutines for asynchronous, non‑blocking IO.
* Event‑driven ingestion via Kafka topics for periodic fare feeds.

## 5.2 Component Diagram

1. Frontend (Next.js) – SSR pages, Tailwind UI.
2. API Gateway (Spring Cloud Gateway) – auth, rate‑limit, routing.
3. Aggregator Service – merges provider results, scoring engine.
4. Provider Adapters – thin clients around each external API.
5. User‑Profile Service – CRUD for accounts + favourites.
6. Redis – cache by search hash (TTL 5 – 15 min).
7. PostgreSQL – persistent data (users, audit, search logs).
8. Kafka – async pipelines for nightly fare dumps / notifications.
9. Prometheus + Grafana – metrics & dashboards.

## 5.3 Deployment Architecture

* Local dev: Docker Compose; each service in its own container.
* CI: GitHub Actions builds images, runs tests, pushes to registry.
* Prod: Google Cloud Run (fully‑managed, low idle cost) or K8s (GKE Autopilot) using free tier credits. Front‑end can live on Vercel (free hobby tier) or Cloud Run as well.
* Terraform provisions all infra.

# 6. Functional Modules

## 6.1 User Management

Registration, login (JWT), password hashing (bcrypt), favourite searches.

## 6.2 Core Business Logic

Flight search, cache lookup, scoring & sorting, itinerary formatting.

## 6.3 Notification System

Optional email alerts (SendGrid free tier) when price drops.

## 6.4 Reporting and Analytics

Grafana dashboards; BigQuery export for ad‑hoc analysis.

# 7. Data Design

## 7.1 Data Model (Simple)

|  |  |
| --- | --- |
| Entity | Key Fields |
| User | id, email, password\_hash, created\_at |
| FlightSearch | id,user\_id?, origin, destination, depart\_at, return\_at, pax, cabin, hash |
| FareOption | id,search\_id,provider, price, currency, segments\_json, baggage\_json, score |

## 7.2 Data Flow

1. HTTP request hits Gateway.
2. Cache lookup → cache miss triggers Aggregator.
3. Aggregator dispatches parallel coroutines to Provider Adapters.
4. Normalise - Merge - Score - Cache - Respond.
5. Selected fare logged for analytics → Redirect.

# 8. Integration Design

* Provider Adapter Interface abstracts REST/SOAP differences.
* Resilience4j circuit breaker & bulkhead per adapter.
* Scheduled Kafka job ingests bulk fare feeds where available.
* OpenAPI spec generated for each internal service; contract tests via Spring Cloud Contract.

# 9. Security Considerations

* TLS 1.3 everywhere; HSTS.
* JWT auth, BCrypt password hashing, refresh tokens.
* OAuth 2.0 support for social login (future).
* OWASP ASVS L1 coverage; Zap baseline scan in CI.
* GDPR compliance: minimise PII, provide delete‑me endpoint.

# 10. Non-Functional Requirements

## 10.1 Performance

* P99 latency: < 500 ms (cache hit), < 2 s (cache miss) for typical 3‑provider search.
* Throughput: sustain 1 k RPS, burst 5 k RPS.

## 10.2 Availability & Reliability

* 99.9 % monthly API uptime.
* Automatic rollback on failed health checks.

## 10.3 Maintainability

* Clean Architecture, 80 % unit test coverage, Detekt + ktlint.
* Dependabot & Renovate for dependency updates.

## 10.4 Security

* Secrets in HashiCorp Vault / GCP Secret Manager.
* SCA via Trivy container scans.

## 10.5 Usability

* Mobile‑first responsive UI, WCAG AA contrast, keyboard nav.

# 11. Technology Stack

| Layer | Tech | Reason |
| --- | --- | --- |
| Back-End | Kotlin 1.9, Spring Boot 3, Coroutine WebClient | Modern, terse; fully OSS |
| Front-End | Next.js 15 (TypeScript), Tailwind, shadcn/ui | SSR for SEO, fast DX |
| D8 | PostgreSQL | OSS, rich SQL |
| Cache | Redis | Low‑latency |
| Messaging | Kafka | Decoupling |
| Infra | Docker, Terraform, Cloud Run / GKE, Vercel | Low‑cost/free tiers |
| CI/CD | GitHub Actions, Kaniko, ArgoCD | End‑to‑end pipeline |
| Testing | JUnit5, MockK, Testcontainers, Playwright, k6 | Shift‑left pyramid |

# 12. Assumptions and Constraints

* All tooling must be FOSS or have a generous free tier.
* No paid flight‑data APIs initially. Use:
  + Amadeus Self‑Service API (2k requests/month free).
  + Kiwi/Tequila API (free key with limited RPS).
  + Travelpayouts (affiliate model) for cheap fares.
* Hosting budget: £0–10/mo using free credits & hobby tiers.
* Since the developer is working solo, strong CI/CD pipelines and automation are essential to reduce manual overhead and ensure project quality.

# 13. Risks and Mitigations

|  |  |  |  |
| --- | --- | --- | --- |
| Risk | Likelihood | Impact | Mitigation |
| Provider API Quota Limits | Medium | High | Cache, multi‑provider fallback, mock data for dev |
| Free tiers revoked | Low | Medium | Abstract infra → swap cloud, local Compose |
| GDPR . data privacy fines | Low | High | Minimal PII, DPA templates, auto‑purge logs |
| Performance degradation under burst load. | Medium | Medium | k6 perf tests in CI, auto‑scaling Cloud Run |

# 14. Future Enhancements

* Hotel/car meta‑search.
* User price‑alert emails & push notifications.
* GraphQL gateway for richer queries.
* CO₂ emissions scoring per itinerary.
* ML‑based personalised ranking.