Distributed Airways

Project: COMP30220 Distributed Systems, Autumn 2021

Ahmed Jouda	Chee Guan (Jason) Tee	Noor Bari	Rajit Banerjee
18329393	18202044	18400034	18202817

Synopsis

Distributed Airways aims to model a metasearch engine for flights. Given a date of travel, source city and destination city, the system presents the user with all the available flight options from multiple airline services. The primary idea is that such a system saves the user the time and effort of exploring different websites for individual airlines, by providing the most cost effective deals from multiple sources. In terms of an architectural overview, the distributed system is designed to allow a client to interact solely with a central broker service, which compiles results from a number of independent airline service providers.

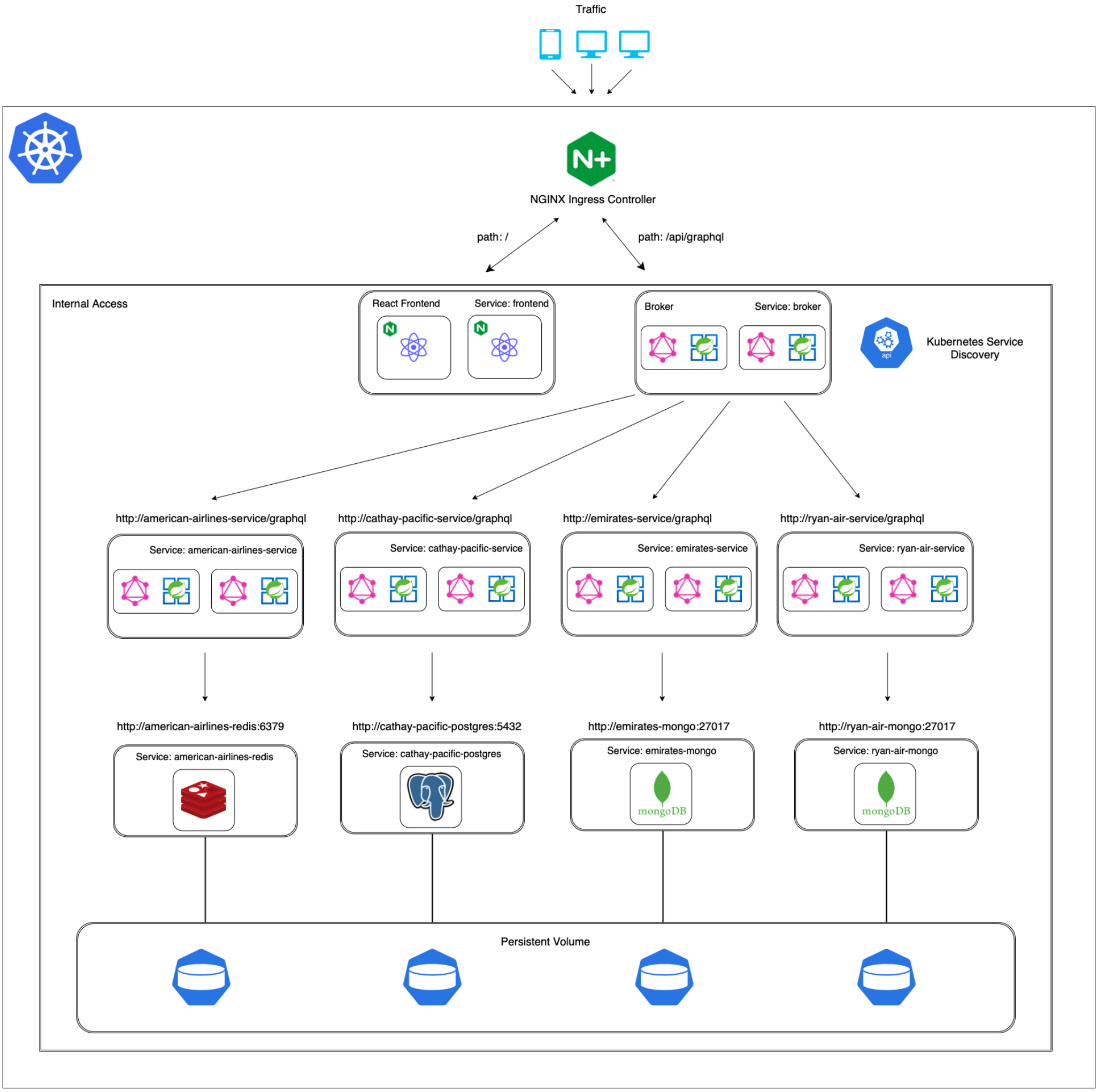
Technology Stack

- Java 8
  - Java is the primary programming language used for backend development.
  - Apache Maven provides build automation and dependency management capabilities.
  - Spring Boot, based on the Spring framework, provides a number of useful abstractions, especially for database connections (Spring Data).
  - Other tools: Project Lombok, Google Guava, GSON, Jsoniter, Jackson, JUnit, etc.
- GraphQL
  - An open source data query and manipulation language, used for the API design of services in the system (an alternative to the REST API architectural style).
- Databases
  - Redis: In-memory key-value store (NoSQL) (used by 1 airline service).
  - PostgreSQL: Extensible and SQL compliant relational database management system (used by 1 airline service).
  - MongoDB: Document oriented database (NoSQL) with optional schemas (used by 2 airline services).
- React.js + TypeScript
  - TypeScript is a strongly typed superset of JavaScript, and React is a component-based library for UI design. They are the primary tools which have been used for frontend development.
- Docker
  - OS-level virtualisation and containerisation tool. Docker Compose has been used for running the multi-container distributed system.
- Kubernetes (K8s):
  - Container orchestration system, which has been used for service discovery, scaling and load balancing.
- NGINX
  - NGINX has been used as a web server for the React frontend, and again as a reverse proxy and load balancer for Kubernetes' Ingress controller (abstracting the complexity of K8s application traffic routing).

System Overview

Component/Module	Overview
`american-airlines/`	Independent airline GraphQL server providing American Airlines flights. Uses Redis as the data store.
`cathay-pacific/`	Independent airline GraphQL server providing Cathay Pacific flights. Uses PostgreSQL as the data store.
`emirates/`	Independent airline GraphQL server providing Emirates flights. Uses MongoDB as the data store.
`ryan-air/`	Independent airline GraphQL server providing Ryanair flights. Uses MongoDB as the data store.
`broker/`	Central service that acts as a GraphQL client compiling flights from the 4 aforementioned airline services, then as a GraphQL server for any type of external client (e.g. CLI client, frontend, or the interactive GraphiQL on a browser).
`core/`	Common classes and utilities re-used by multiple services.

`cli-client/`	Simple command line interface client to interact with the broker service and retrieve flights for fixed search parameters.
`ui/`	React and TypeScript based frontend user interface to interact with the broker service backend and display results.
`k8s/`	Service, deployment, persistent volume, ingress and cluster configurations for Kubernetes.



- The system follows the principles of microservice architecture, with loose coupling between the different services.
    - This allows the freedom of choice regarding individual service design, e.g. the database and frameworks used (or even the programming language if desired).
    - As long as every service adheres to a pre-defined GraphQL schema, the low-level details of how and where the data fetched from is left up to the developers.
- The first gatekeeper that external clients hit is the NGINX Ingress controller (reverse proxy/load balancer) that manages access to the Kubernetes cluster of containerised applications.
    - The root path directs the client to the React frontend, which is a website being served using NGINX (as a web server).
- The frontend accesses the `/graphql` endpoint of the central broker service through the Ingress, and sends a query based on user input on the web interface (selection of travel date, and source/destination city).

- The central broker service identifies the running airline services using K8s service discovery, and uses GraphQL queries to retrieve flights from each airline, satisfying the parameters that the broker itself received from a client request (date of travel, source city and destination city). The results are compiled and returned to the client. Two other utility queries are also supported (to be used by the frontend client) to retrieve the list of all supported source and destination cities from each airline.
- It's worth noting that each service in Kubernetes is an internal load balancer which routes traffic to individual pods/containers and can be scaled easily depending on the traffic/load.
- Each airline service is also connected to a database (Redis, PostgreSQL, MongoDB using Spring Data abstractions). The use of K8s persistent volumes for data stores provides a life-cycle that is independent from any Pod (smallest, most basic deployable object) that is using it. In other words, if a database container fails, we can easily spin up another one without any data loss.
- Containerisation with Docker, and the use of Kubernetes makes scalability and fault tolerance integral parts of the distributed system design.

Contributions

Every team member contributed at least one airline module (including API design, database setup and Dockerisation) in the multi-module Maven project.

- **Ahmed Jouda**
  - Airline module: `ryan-air/`
  - Integration with database: MongoDB
  - Flights data collection
  - Documentation
- **Jason Tee**
  - Airline module: `american-airlines/`
  - Integration with database: Redis
  - Scaling, load balancing and service discovery using Kubernetes
  - Central module: `broker/`
  - Architecture diagram
- **Noor Bari**
  - Airline module: `cathay-pacific/`
  - Integration with database: PostgreSQL
  - Documentation
- **Rajit Banerjee**
  - Airline module: `emirates/`
  - Integration with database: MongoDB
  - Central modules: `core/`, `broker/` and `cli-client/`
  - Frontend module: `ui/`
  - Documentation

Reflections

A number of challenges were overcome during the course of the project, which proved to be a great learning experience. If we had to start again, we would tackle a number of these issues differently.

- Version and compatibility issues for Maven dependencies.
  - It was quite challenging to determine the correct versions for Spring Boot parent, Spring Cloud, and Spring Data related dependencies that would be compatible with each other, due to inadequate documentation.
- Service discovery
  - To avoid hardcoding the hosts and ports for running airline services, a Netflix Eureka-based solution was implemented initially. As a load balancer and service discovery tool, it was relatively simple to implement a Eureka service registry to keep track of connected services. However, upon realising that service discovery was a built-in feature in Kubernetes, we decided to abandon the Eureka approach.
- Cassandra
  - Apache Cassandra was initially considered as another option for an airline data store, however, persistent dependency issues with Spring Boot forced us to switch to the more user-friendly MongoDB.
- PostgreSQL
  - Strictness with case sensitivity in PostgreSQL led to a number of errors initially. This was overcome through research and eventually homogenising column names to make them all lowercase instead of the more popular camel case.
  - Additional configurations were required to accommodate list-type fields in the database.
  - There were a lot of versioning issues in the pom file. It was difficult to find the correct versions of dependencies to get the database working properly.
- Data storage and JSON structure decisions
  - It was slightly challenging to decide on a consistent structure for flight data stored as JSON files (for simplicity, we avoided externally hosted databases).

    - Transit airports and non-direct flights representation: Certain attributes such as flight number had to be represented as an array (for 2 separate flights).
    - Different categories of tickets: Each airline offers different ticket categories with varying prices. To overcome this, each airline service described its own categories in the common `category` field and adjusted the price accordingly.

Much of the technology stack we used was relatively new to us, and building a full-fledged distributed system taught us about the various limitations and benefits of the tools.

- Java, Maven, Spring Boot, Spring Data (+ databases)
    - Benefits
        - The use of Spring Boot with Java (+ Lombok) greatly reduced the boilerplate code required to get the services up and running.
        - Spring Data took care of database connections, and the required code changes were minimal even with a variety of different databases such as Redis, PostgreSQL and MongoDB.
    - Limitations
        - Maven dependency issues related to Spring were quite troublesome.
        - Even with Spring Data abstractions, there were some nuances involved with model and repository setup for databases, which were not well documented online.
- GraphQL
    - Benefits
        - Growing popularity and user community.
        - Compared to REST (server-driven), GraphQL is client-driven.
        - Only a single API endpoint (`/graphql`).
        - Auto-generated API documentation.
        - Faster: avoids over-fetching or under-fetching data thanks to predefined schemas and query shapes.
        - More stable than REST due to built-in validation and type checking.
    - Limitations
        - Learning curve.
        - Lack of control over caching, rate limiting and handling complex queries (extremely nested queries).
- React/TypeScript
    - Benefits
        - Extremely useful (and popular) reusable component libraries available.
        - Large community of users and extensive documentation
    - Limitations
        - Learning curve.
- Docker, K8s, NGINX
    - Benefits
        - Containerisation is a vital part of microservice design.
        - Service discovery, scaling and fault tolerance with Kubernetes can be handled simply by setting up a few YAML configuration files.
    - Limitations
        - Multi-architecture builds (supporting both arm64 and amd64) for Docker images were time consuming and required the use of an experimental `docker buildx` feature.
        - Steep learning curve for Kubernetes.