Design for Online Airline Reservation System

# Core pattern

Microservice architecture is chosen for the online airline reservation system. Firstly, the system built as microservices can be divided into multiple component services. Unlike the monolithic architecture, the microservice architecture can be organized around business capabilities and priorities. Also with agility and scalability.

NOTIFYING SERVICE

API GATEWAY

TICKET RESERVATION

SERVICE

FLIGHT COMPARISON

SERVICE

SEARCHING TICKET

SERVICE

CUSTOMER LOGIN

SERVICE

RATING SERVICE

PAYMENT SERVICE

FEEDBACKDB

PAYMENTDB

ACCOUNTDB

RESERVATIONDB

FLIGHTDB

TICKETDB

# Decomposition

Decomposing by business capability is chosen for this system. The services in the system can be identified by their business purpose. For examples,

* Searching tickets and flight comparison correspond to Inventory management.
* Ticket reservation corresponds to reservation management.
* Customer login corresponds to user management.
* Payment via credit card corresponds to payment management.
* Rating corresponds to feedback management.
* Notifying corresponds to notifying management.

# Deployment pattern

Serverless deployment is chosen for this system. Using this deployment infrastructure, any concepts of servers can be hide, like physical or virtual hosts, or containers. Packaging the code and upload it to the deployment infrastructure to deply the services. It can be faster and make you can focus on the code. AWS Lambda has the richest feature set. AWS Lambda functions are stateless components that is invoked to handle events. They can response to any events generated by AWS services like S3 or DynamoDB.

AWS

LAMBDA

ACCOUNT

DB

PAYMENT

SERVICE

# Cross cutting concerns

Microservice chassis framework is chosen for this system to handle cross-cutting concerns. In order to create a new microservice, cross-cutting concerns such as externalized configuration, logging, health checks etc. must be handled. Microservice chassis for each programming language/framework build can make developing microservices quicker and easier.

# Communication pattern style

Remote procedure invocation (RPI) is chosen for this system. In this system, services must handle requests from clients. Sometimes the services may collaborate to handle those requests. RPI can be used for inter-service communication. Means it lets clients use a requests/reply-based protocol to make requests. It can be simple and familiar.

# External API

API gateway is chosen for this system to let clients to access the individual services. Precisely, Amazon API Gateway can be used to manage services. Amazon API Gateway can handle all the request and process up to concurrent API calls, including traffic management, authorization and access control etc. It will help with locating the right services for hundreds or thousands of request from clients. And it can work along with other Amazon services like Amazon EC2 or AWS Lambda.

# Service discovery

Server-side service discovery is chosen for this system. It can help to address the requests from clients to the right service instance. Client makes requests via a router (load balancer) that runs at a well known location. And the router will forward the requests to an available service instance. AWS Elastic Load Balancer (ELB) can be used as the router. It can handle the varying load of client requests in availability zones. It provides highly availability and salability.

# Reliability

Circuit breaker is chosen for this system to prevent failure from cascading to other services. In microservice architecture, there is always the possibility the one service is unavailable when the other service synchronously invokes. When failures happen, the circuit breaker trips, and all attempts to invoke the services will fall immediately. After a timeout, only a few test request can pass through. Only the test requests success, the circuit breaker resumes normal operation.

# Data management

Database per service is chosen for this system. In microservice architecture, services are loosely coupled. So the databases are needed for each service and can only via by API. DynamoDB from AWS can be used as NoSQL database for this system. Compared with relational database, NoSQL database is good at storing complex and unstructured data. Also, it can trigger the AWS Lambda function.

# Security

Access token is chosen for this system to verify the authorization to perform an operation. AWS Identity and Access Management (IAM) can be used and it associates with other services provided by AWS.

# Testing

Service component test is chosen for this system. To avoid end to end testing with is difficult, slow and expensive. Isolating the service and test it can be easier and faster to test the service to meet the requirement. However, test may pass but the application may fail in production. So service integration contract test is also needed to test the service meets the consuming service’s expectations.

# Observability

Log aggregation is chosen for this system. Amazon CloudWatch can be used to monitoring services on AWS cloud used in application. Metrics and log files can be collected and monitored. It can help to understand the behavior of an application and troubleshoot problems.

# UI pattern

Client-side UI composition is chosen for this system. The client-side UI is needed for displaying data or user interacting. It can be written in HTML, JS etc. The resources can be stored in Amazon S3.