Saibot International Airport

Motivational Document

Sven Westerlaken
Maikel Jacobs
Mats Gemmeke
Twan van Maastricht
Luka Brinkman
Thimo Koolen

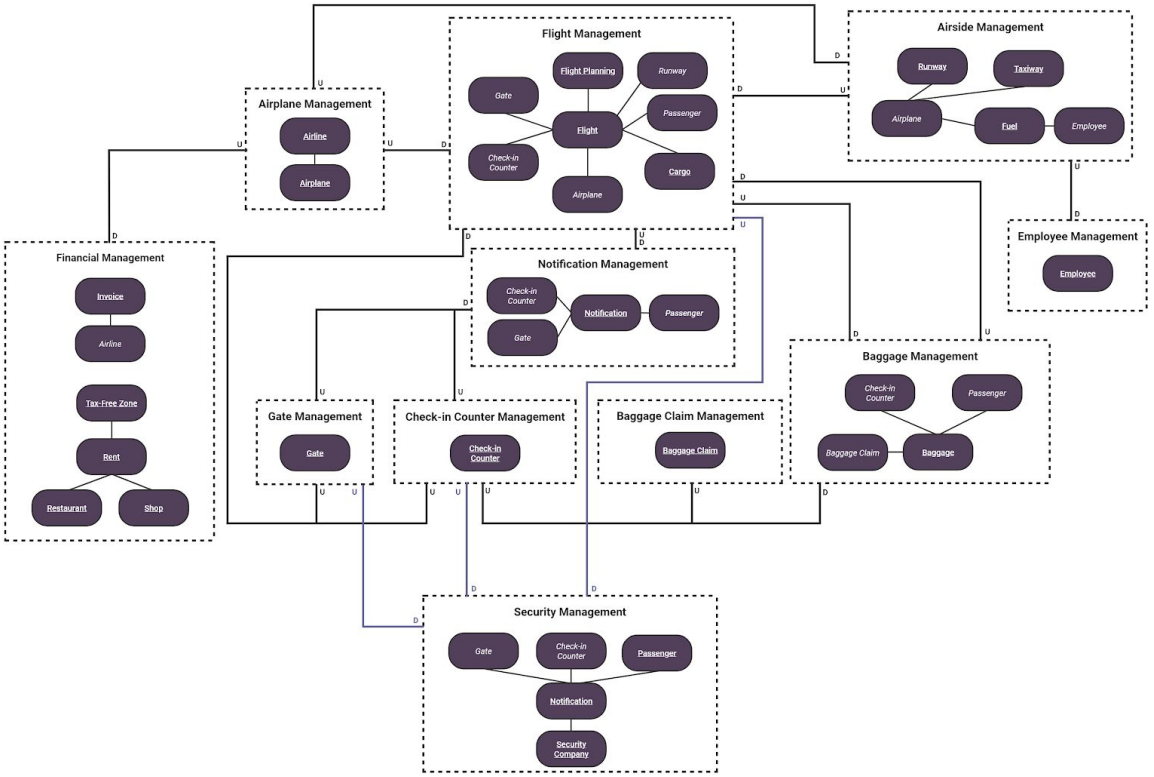
Architectural Concepts

Microservices based on the principles of DDD

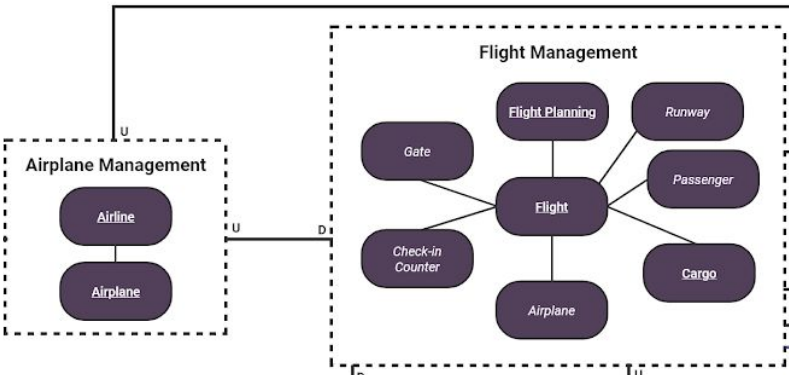
Het project is begonnen met het ontwikkelen van een Context Diagram op basis van DDD (Domain Driven Development). Dit diagram is hieronder te vinden in afbeelding 1.

Het core domain bevindt zich in de Flight Management microservice. Hier komen alle subdomains samen, waaronder de Airside Management, Airplane Management en Gate Management subdomeinen. Het domain is de airport en de subdomains zijn de bijbehorende processen binnen de luchthaven.

De diverse subdomains zitten elk in hun eigen bounded context. Binnen elke bounded context zitten diverse classes, de domain objects. Onderlinge communicatie tussen de verschillende bounded contexts gebeurd door middel van messaging. Voor meer informatie hierover, zie het kopje *'Event driven architecture based on messaging'*.



Afbeelding 1: Context Diagram Saibot Intl. Airport



Afbeelding 2: Bounded contexts met upstream en downstream

In afbeelding 2 zijn de aggregrate roots en aggregrates te herkennen. In de Airplane Management bounded context is ‘Airplane’ de aggregrate root met Airline als bijhorende aggregrate. De context dat data produceert (de producer) wordt aangegeven met ‘upstream’, de source of truth, en de context dat de data consumeert (de consumer) wordt aangegeven met ‘downstream’. Dit houdt in dat de Flight Management context data uit de Airplane Management context gebruikt doormiddel van messaging.

Eventual Consistency

Eventual Consistency is toegepast binnen de Airside Management microservice. Deze microservice is verantwoordelijk voor de verschillende runways waarop vliegtuigen landen en opstijgen, de taxiways om vliegtuigen van de runways naar de gates te laten taxiën (en vice versa) en de fuel tanks, waarbij kerosine in de vliegtuigen gepompt kan worden.

Eventual consistency ontstaat omdat de gegevens niet direct consistent zijn. Een systeem kan ofwel consistent zijn, ofwel available (CAP-theorem). Omdat een hoge availability belangrijk is voor het systeem (de luchthaven-operatie gaat gewoon door), is er gekozen om eventual consistency toe te passen. Gegevens op de Airside Management worden wanneer mogelijk consistent gemaakt met andere microservices. Een voorbeeld zoals in het Context Diagram te zien is, is het domain model *Runway* zowel in de Airside Management gebruikt als in het Flight Management. Aanpassingen aan Runway-data in Airside Management moeten uiteindelijk ook in het Flight Management aanwezig zijn.

De aangepaste Runway-data gaat als event naar de message broker, in dit geval RabbitMQ. Dit bericht wordt met een bepaald ‘topic’ naar de Exchange gestuurd, waarna alle bekende queues die deze topics uitlezen het bericht ontvangen. In het bovenstaande voorbeeld zal Flight Management bijvoorbeeld een bericht op topic *runway.create* krijgen, waarin de data van de runway staat. Het Flight Management kan hier zelf iets mee doen.

Als de Flight Management module offline is (of de verbinding is verbroken) op het moment dat een nieuwe runway aangemaakt wordt, blijft het bericht in de queue staan. Immers is de queue al bekend binnen de message broker. Als de verbinding hersteld is, zal het event alsnog afgeleverd worden bij Flight Management, waarna de data weer consistent is.

De message broker slaat alle events ook persistent op. Dus data zal ook een crash of restart van de message broker overleven. Uiteraard zijn er altijd use cases waarop deze data verloren kan gaan, zoals een wipe van de harde schijf. Dit is op diverse manieren op te lossen, maar valt buiten de scope van eventual consistency. Data kan nooit 100% veilig en altijd bereikbaar zijn.

Als de message broker niet bereikbaar is, zal Airside Management het event zelf in de eigen database opslaan. Er wordt regelmatig geprobeerd om opnieuw te connecten met de message broker. Wanneer dit gelukt is, worden alle events alsnog naar de broker gestuurd en uit de database verwijderd. Voor het herstellen van de events wordt een aparte channel opgezet, omdat dit proces geen invloed mag hebben op de afhandeling van nieuwe events. Dit zou kunnen voorkomen wanneer de broker voor een lange tijd niet beschikbaar is geweest, waardoor een grote hoeveelheid aan events hersteld zal moeten worden. Het opgezette channel wordt na afloop weer afgesloten. Consumers vanuit Airside Management zullen evenals worden hersteld om het verwerken van events weer mogelijk te maken.

Event driven architecture based on messaging

Binnen het applicatielandschap van Saibot wordt gebruik gemaakt van event driven architecture. Dit staat in contrast met het bekende request driven architecture, waarin er requests tussen diverse applicaties gedaan worden en er dus een sterke mate van tight coupling bestaat. Event driven architecture maakt gebruik van messaging op een message broker (zoals RabbitMQ), waar een bepaalde actie op verstuurd wordt. Andere applicaties die geïnteresseerd zijn in het soort bericht, kunnen hier zelf acties op uitvoeren.

Door het gebruik van een event driven architecture, is de mate van loose coupling tussen de verschillende applicaties in het applicatielandschap versterkt. Hierdoor is flexibiliteit makkelijker, omdat een andere applicatie simpelweg het event moet kunnen ontvangen en verwerken met zijn eigen business logic. Ook is een applicatie beter schaalbaar, omdat de verschillende applicaties allemaal los de events uitlezen via de message broker.

Command Query Responsibility Segregation (CQRS)

Kort uitgelegd is CQRS volgens Fowler(2011): “The notion that you can use a different model to update information than the model you use to read information”. Dit is onder andere toegepast bij de uitwisseling van data tussen de Airplane Management service en de Flight Management service. De dataobjecten uit de Airplane Management worden namelijk door middel van messages doorgestuurd naar de Flight Management (bijvoorbeeld bij een POST request) en in deze service in een better to read vorm opgeslagen. De Airplane Managment

maakt gebruikt van MySQL, terwijl de Flight Management gebruik maakt van MongoDB. Doordat de data uit de MySQL database in documents (Mongo) wordt opgeslagen (voor bijvoorbeeld flights, airplanes gates en overige domein objecten), maakt het de data makkelijker om uit te lezen door middel van subdocumenten. Kortom, geoptimaliseerd voor het lezen in plaats van het schrijven.

Event Sourcing

“Event Sourcing ensures that all changes to application state are stored as a sequence of events.” (Fowler, 2016). In onze applicatie is dit toegepast in de security management. Het idee was eerst om dit in een aparte microservice aan te bieden, maar door tijdgebrek is deze functionaliteit samengevoegd met de requirement dat alle events uitgelezen moeten kunnen worden door de security. Het is nu mogelijk om alle events via een http request op te vragen in zowel event (json) formaat als log (strings) formaat. Hierbij dient het log formaat als het inzicht voor de security en het event formaat om eventueel meer informatie op te halen of juist voor event sourcing.

Het is namelijk mogelijk om te filteren op topic/aggregate. Op die manier kan hierover alle events opgehaald worden. Het is vervolgens aan de vragende microservice om deze events één voor één te laten afspelen. Op die manier kan de huidige state van de opgevraagde aggregate bepaald worden. Echter is op te merken dat het laatste gedeelte van deze logica nog niet concreet is uitgewerkt vanwege tijdnood. Daarentegen is dit eenvoudig te implementeren door het data-component in het event object met de andere te vergelijken aan de hand van de timestamps (volgorde van events). De overgebleven data kan dan naar behoren opnieuw gepersisteerd/verwerkt worden.

Enterprise Integration Patterns

1. Toepassing door middel van RabbitMQ. Wij hebben gekozen voor Topic Exchange om messages te produceren en te consumeren. Aan de producer kant wordt een message verstuurd met een routing key naar de topic exchange, en aan de hand van een match tussen routing key en de binding routing patterns van alle queues worden de berichten weer geconsumeerd. Dit komt overeen met de Enterprise Integration Pattern ‘Message Filter’ waarbij op basis van een set criteria (de topics) de berichten worden gefilterd en zo wordt geselecteerd welke berichten de consumer wel en niet wilt ontvangen.
2. Toepassing van Enterprise Integration Patterns door middel van Camel. Passengers melden zich aan bij een airline. Dit gebeurt dus buiten ons systeem om. De microservice flight management heeft passagierdata nodig om vluchten te kunnen beheren. Ook kan het voorkomen dat airlines hun passagierdata in verschillende vormen opslaan. Dan moet binnenkomende data dus eerst getransformeerd worden naar een door ons bruikbaar model (canonical data model). Systemen van externe airlines moeten dus geïntegreerd worden met ons systeem. Dit wordt gedaan aan de hand van een camel applicatie. Camel maakt een Consumer en een Producer aan

aan beide kanten van een message channel (of meerdere Consumers naar één producer met meerdere point-to-point channels wanneer er sprake is van meerdere airlines die passengers opslaan). De consumers vragen in intervallen data op uit de databases van de airlines. Hierna wordt deze data getransformeerd en omgezet naar json. Vervolgens wordt de data doorgezet naar de producer die passagierdata post naar een REST endpoint in de microservice flight management waarin de data gerepliceerd wordt in een read-model.

Helaas is het niet gelukt om al het bovenstaande te kunnen implementeren (althans, niet alles werkt naar toebehoren).

.

Non-functional Requirements

- Events should not be lost within the airport.
- The application should be able to work on a consistent speed, even in case of an unexpected or unusual high load.
- The application should still function when one component fails, except for said component.
- Failed components should restart as soon as possible.
- Connections to the message broker should be restored when possible.
- Data should not be lost when connection to the message broker is lost.
- Data should be send to the message broker when a connection is established again.
- Service should only be unavailable because of its own reasons.
- Where required due to restrictions, events should be backed up to reconstruct past states.
- Services should be open to modification.
- Services should be extensible and minimally affected by new functionality.
- Each service should be deployable on its own.
- Each service should be the owner of its data and thus be the only one able to modify it.
- Services should be able to access data from other services only through events emitted after creation, modification or deletion of data on the primary service.
- Documentation is available to provide detailed information about the implementation of certain concepts.
- Services should support both horizontal and vertical scaling.
- Data models should be constructed to accommodate the user’s needs.
- Response times should be as short as possible.

Functional Requirements

Requirements marked in bold are implemented in the current version. Other requirements are scheduled to arrive within the software in the next three major releases.

- **An airline is able to register itself at an airport.**
- **An airline can park their planes at a gate.**
- **An airline can arrive and depart with their planes at scheduled runways.**
- **An airline can request time slots for flights.**
- **An airline can register airplanes.**
- **The airport can schedule flights requested by airlines.**
- **The airport can assign a gate and check-in counters for each flight.**
- Passengers are informed of gate and check-in counter changes for their flight.
- Passengers are able to check themselves in at the check-in counter.
- Visitors are able to check in their baggage at the check-in counter.
- All airport customers are billed through the airport financial department.
- All payments have to be successfully authorised before finalizing an action or request.
- Visitors are able to park at the airport.
- All internal transactions in the airport are also billed through the airport financial department.
- **Security is able to have insight in all airport events.**
- External baggage companies are notified of changes in baggage status.
- External baggage companies are able to notify airport of changes in baggage status.
- Passengers are able to claim their baggage
- Plane is not able to leave the gate until authorized by ground personnel and control tower.
- A plane cannot land or take-off on runway without authorization of the control tower.
- Airside department notifies ground personnel when and where planes need to be refueled.
- Retail is able to rent a spot at the tax-free zones.