# Airline Ticket Reservation

## Introduction

We're being requested by a customer to create the Architecture and Design of an Airline Ticket Reservation and then implement it's framework.

## System Requirements

The requirements for our airline ticket reservation services include:

- User Management
  - Public users and air line staff users
- Social Login: Google+, Twitter, or LinkedIn
- Reservation Management
- Payment Management
- Flight Management
- Email Alerts
- API Based

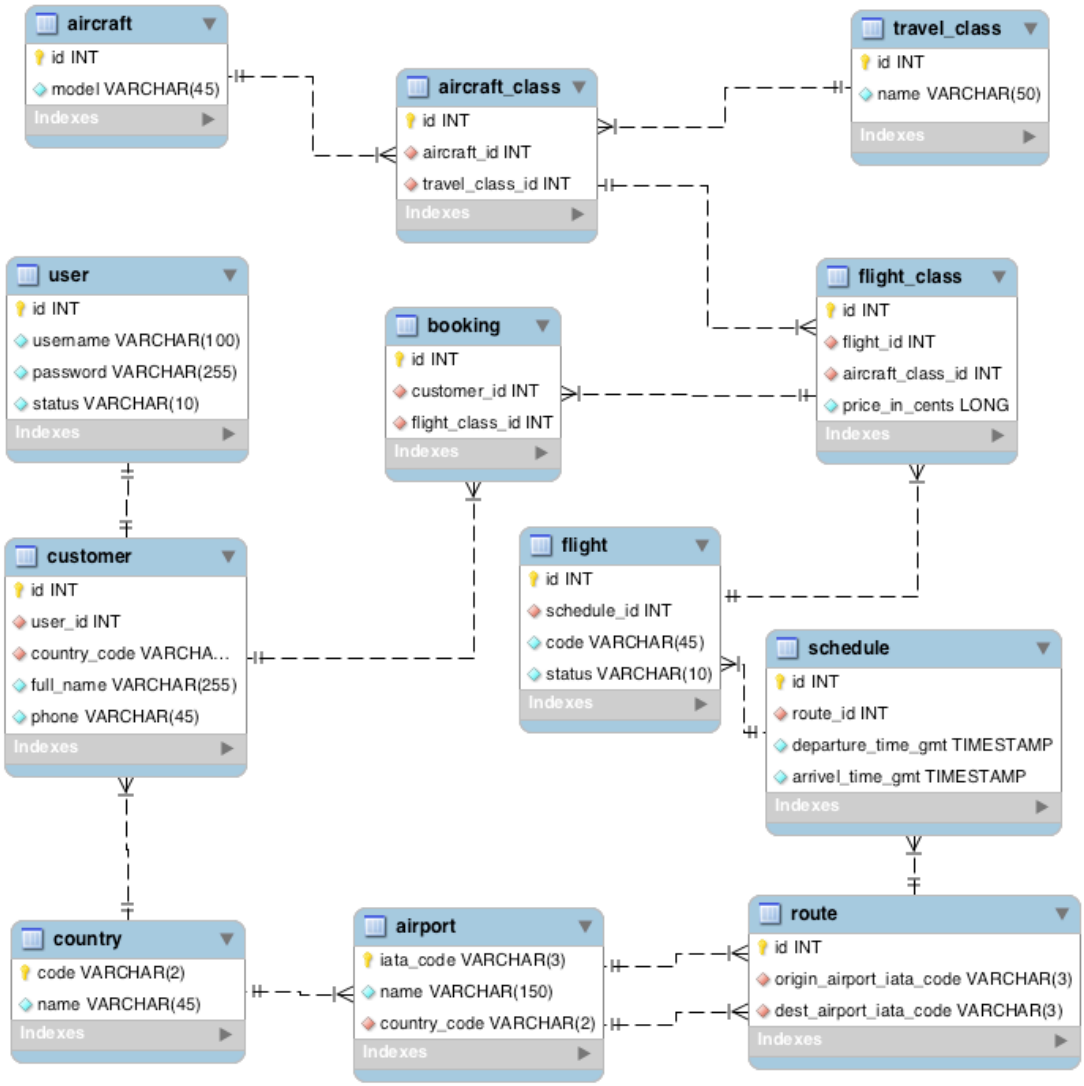
From the high-level requirement services we'll be able to identify detailed services in the following sections of this document.

## Data Model

In this section we'll see a high-level presentation of the data model.

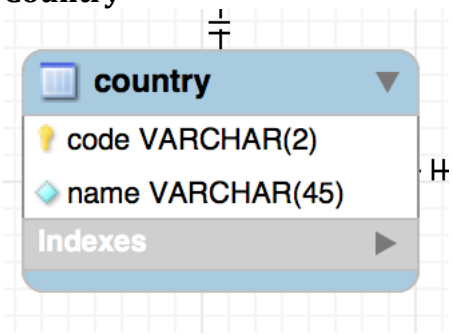
Due the nature of the business we'll be using a relational database to store the data of the entire system.

## Data Model Overview



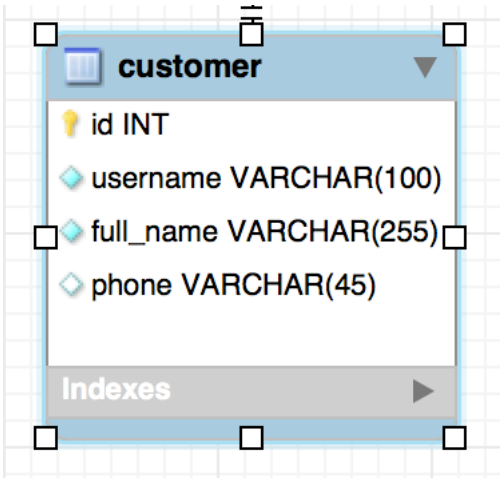

## Entity Description

### Country



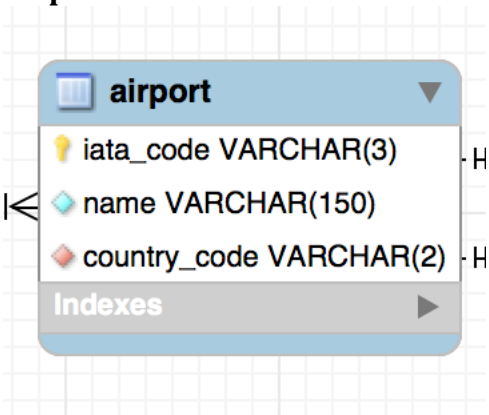

This entity will store the country data and it's code will be the primary key. The attributes are code and name.

**Customer**



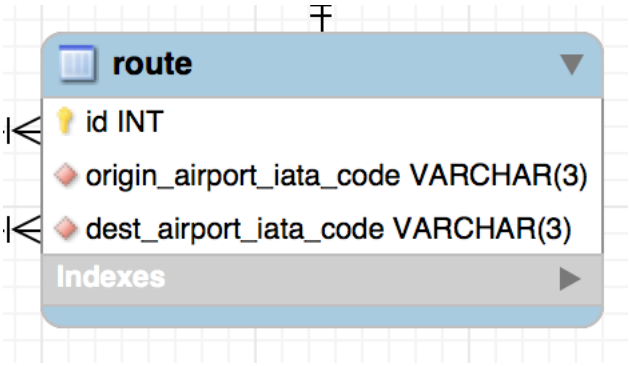

This entity will store the customer data.
The attributes are id, username, full_name, phone.

**Airport**



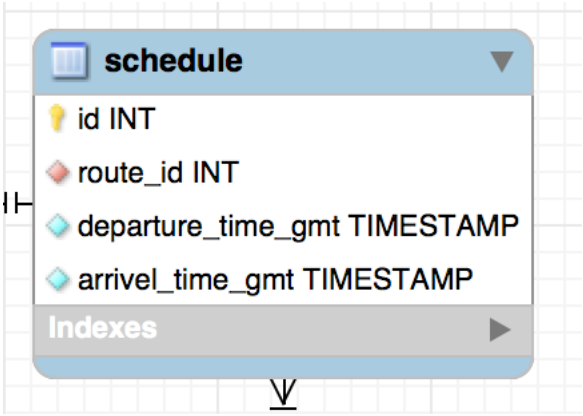

This entity will store the airport data and it's country
The attributes are id, name and country_code.

**Route**



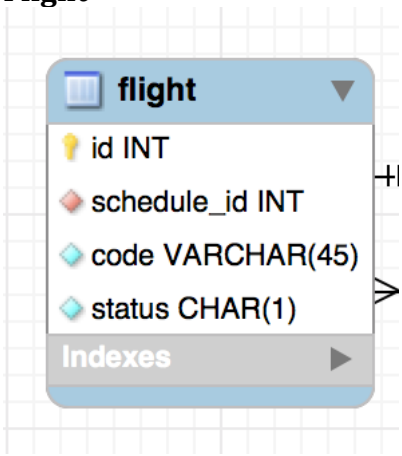

This entity will store the routes of the flight's schedule making a link between two airports.
The attributes are id, origin_airport_iata_code and dest_airport_iata_code.

**Schedule**



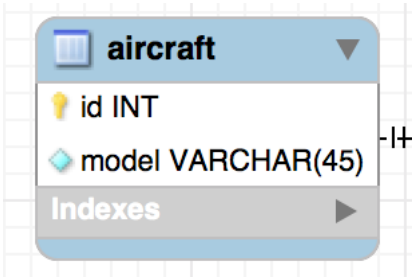

This entity will store the schedule for the flights making a like to the route.
The attributes are id, route_id, departure_time_gmt and arrivel_time_gmt.

**Flight**



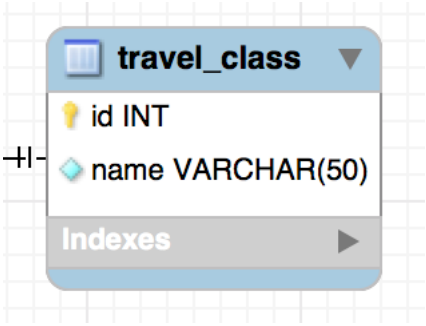

This entity will store the flight data making a link to schedule.
The attributes are id, schedule_id, code and status.

**Aircraft**



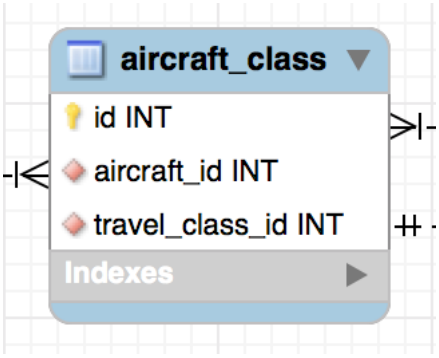

This entity will store the aircraft data.
The attributes are id and model.

**Travel Class**



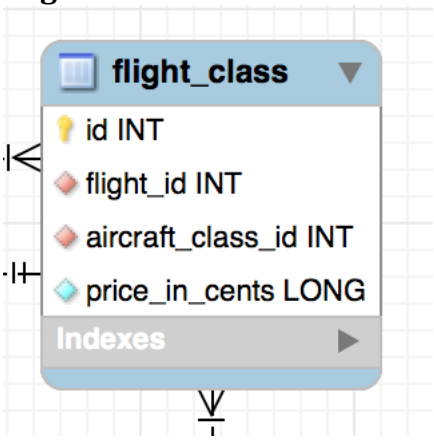

This entity will store the travel class data.
The attributes are id and name

**Aircraft class**



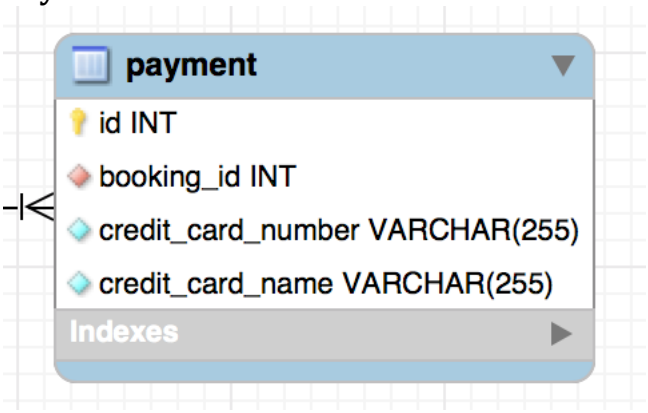

This entity will store the data of the class of an aircraft and a travel class.
The attributes are id, aircraft_id and travel_class_id.

**Flight class**



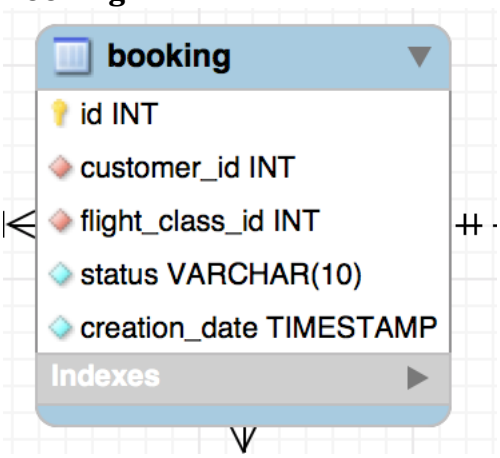

This entity will store the price of a class for given flight.
The attributes are id, flight_id, aircraft_class_id and price_in_cents.

**Payment**




This entity will store the payment data for a booking.
The attributes are id, booking_id, credit_card_number, credit_card_name.
***Note:*** Those cards should NOT be stored in a real system, it should attempt to some KPI

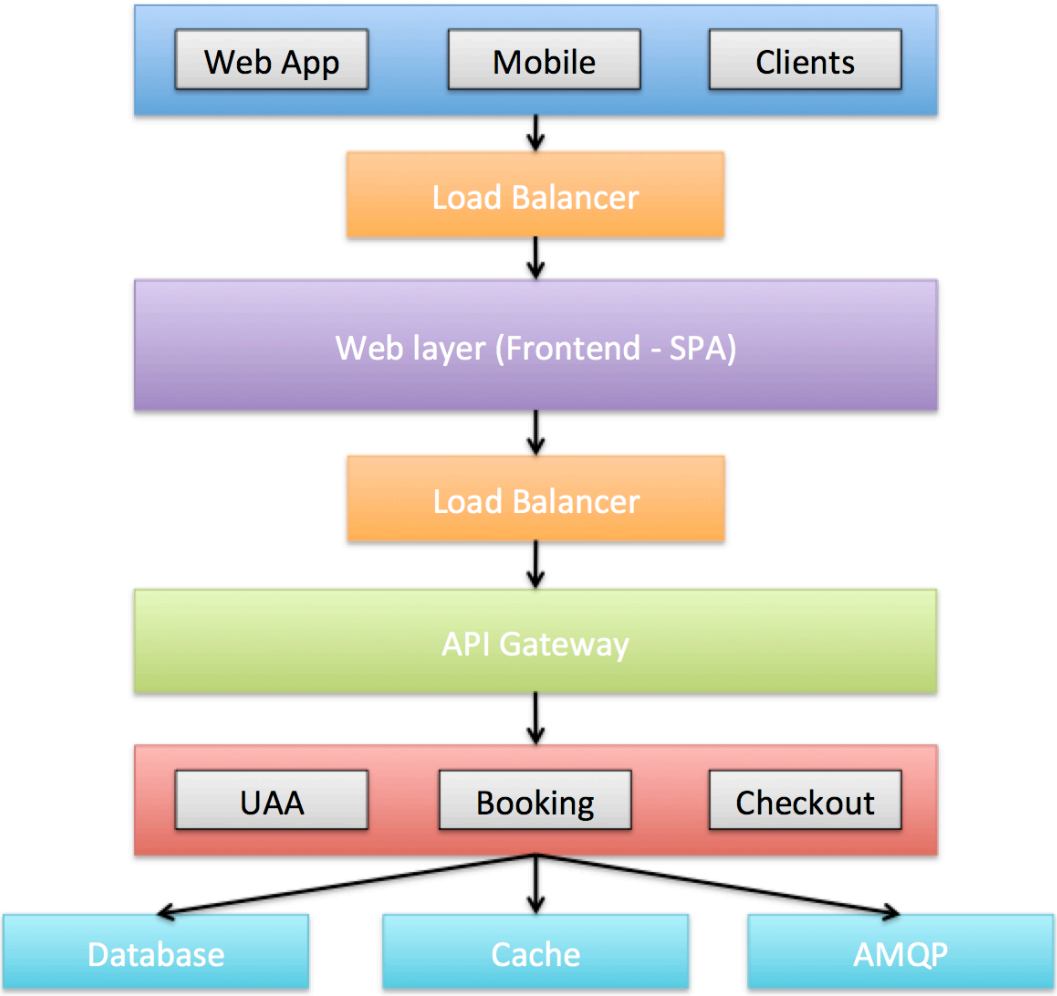
**Booking**




This entity will store the booking data for given customer and flight.
The attributes are id, customer_id and flight_class_id, status, creation_date.

## Architecture Blueprint

In this section we'll cover a high-level architecture components.




In this blue print we can find all layers of our airline ticket reservation marked by different colors.

- Dark Blue: Contains any of our clients that will be consuming our API to build it's interface, it can be webapps, mobile or anyone possible to consume our API
- Orange: Represents a Load Balancer
- Purple: A cluster of our Single Page App
- Green: An API Gateway responsible to secure and route the requests to the proper service.
- Red: Backend layer containing all web services to build the system's API.
- Light Blue: Any external dependency of our backend apps.

## Detailed Architecture

Due the system requirements we have made a choice to use the architectural style of microservices due it's scalability and high availability benefits.

### 1. Microservices requirements

To build a microservice architecture there are few requirements that cannot be forgotten and we going to walk through them in this section.

**Definition:**

*In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.*

*-- James Lewis and Martin Fowler*

With that in mind we'll meet the microservices patterns.

#### 1.1 Service Discovery

Once we're playing with a constellation of services and volatile machines we should use a client-side service registration. It means that the service will register in a Service Discovery by itself.

#### 1.2 Distributed Configuration

One of the pains of a distributed system is to make system configuration governance, and according to 12factor manifesto any configuration should be outside of the application making it possible to run the same build in any environment.

#### 1.3 API Gateway

The API Gateway pattern comes in hand to make a distributed system have a single entry point making it become like a single applications for the consumers.

#### 1.4 Security

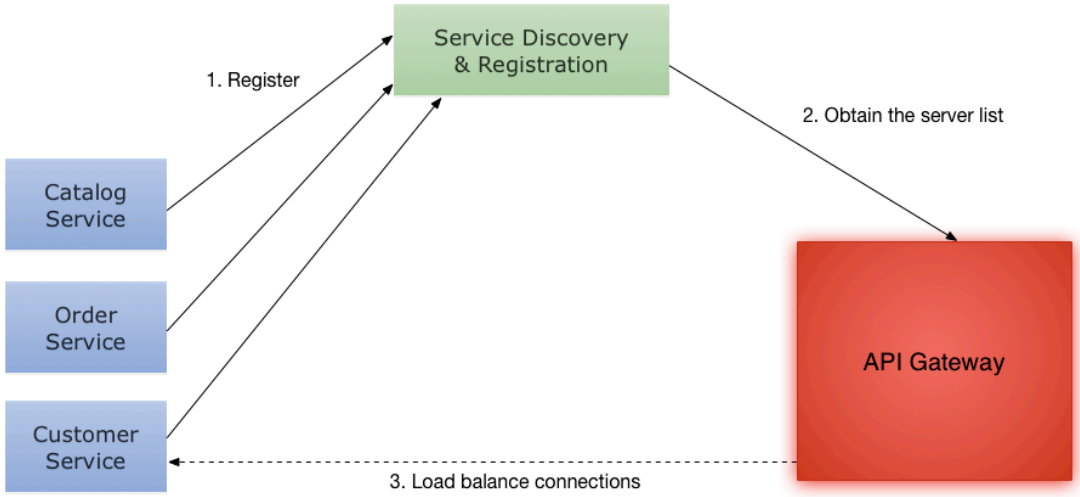
Security in any system is not an easy job, in distributed system it becomes even harder. A common approach to use is secure the APIs with OAuth2 and a centralized Auth Server, with bearer tokens or JWT tokens.

## 2. Technologies

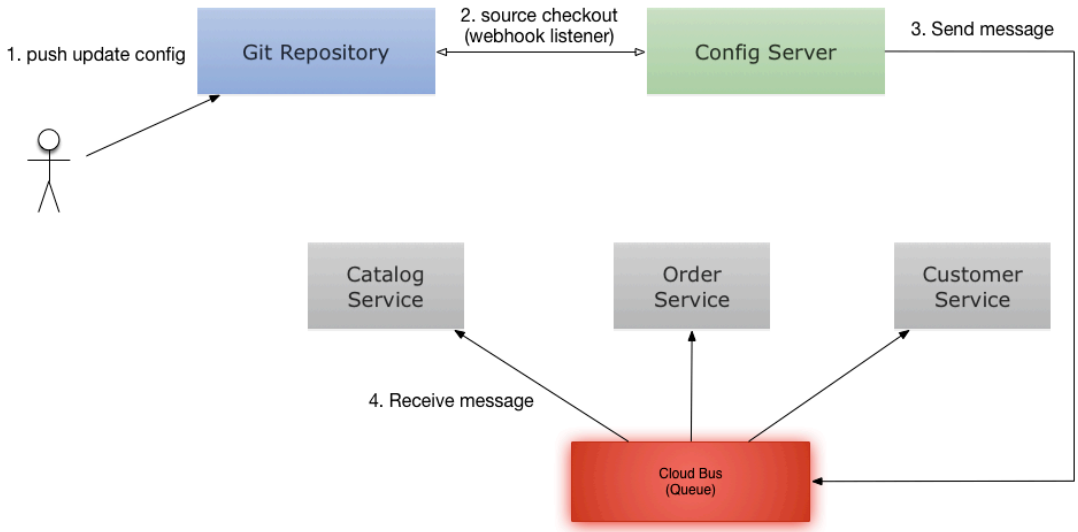
To build this project and cover all microservices requirement I decided to use the Spring Cloud Netflix (OSS) framework:

- Service Discovery: Netflix Eureka
- Distributed Configuration: Spring Cloud Config
- API Gateway: Netflix Zuul
- Security: Spring Security OAuth

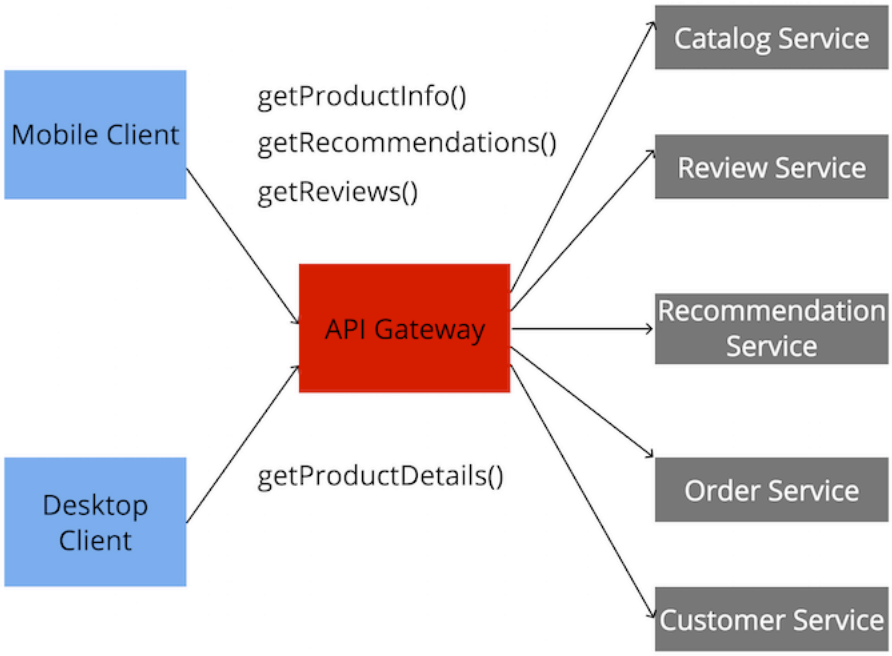
### 2.1 Service Discovery Flow




### 2.2 Distributed Configuration Flow



### 2.3 API Gateway Flow




## Breakdown the system components

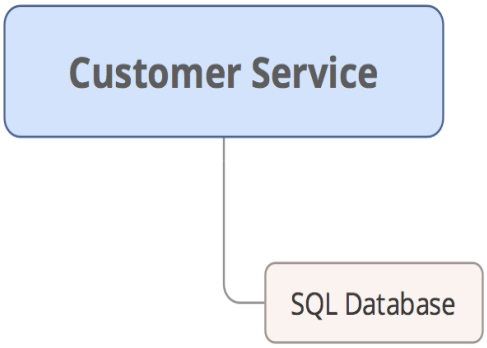
Once we know the frameworks and the architectural style chosen to build this system, now we can breakdown the system components into a detailed view.

This system will be divided in few services that are:

- Customer Service
- Booking Service
- Search Service

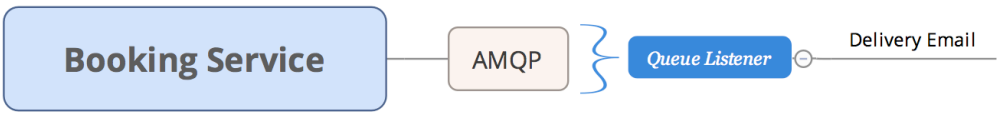
It could be divided in few more services but for a start it will fit.

- **Customer Service**
  The customer service will be responsible to manage customers making operations like CRUD operations on a database.



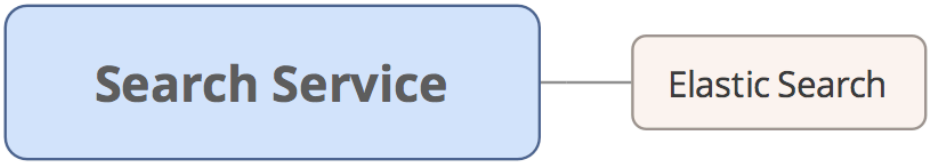

The customer service will be dependent only of a SQL database to make his CRUD operations.

- **Booking Service**
  The booking service will be responsible to manage CRU operations of a booking. The booking table on database will never delete a row, it just updates it status. The values that can be: CONFIRMED, CANCELED or PAID.




All operations won't be performed directly to the database, it should be persisted in a queue and a listener should operate the database operations and trigger the email alert to the customer.

- **Search Service**
  The search service will be responsible to manage all search operations in the whole system. It will never reach the database directly, it always lookup information from a search engine, for our use case it will be Elastic Search.

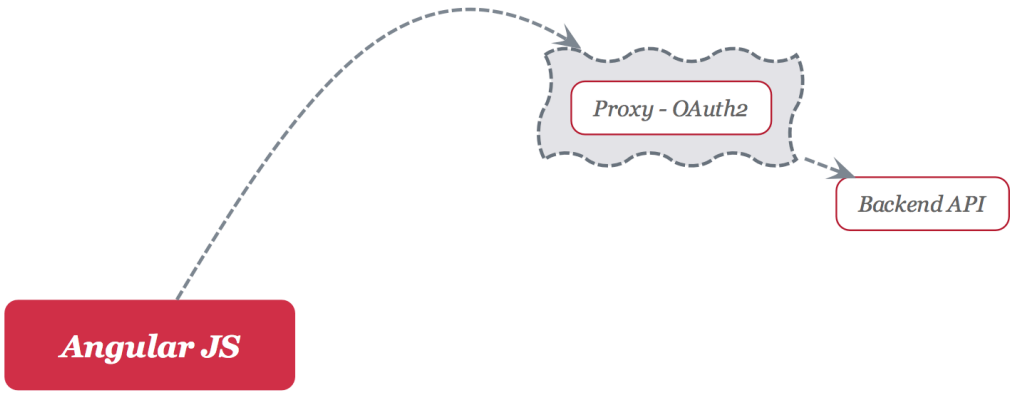



The elastic search will be populated by another batch system, the search service won't be aware of it.

Once we have detailed our backend services we can now go through the frontend customer interface. One of the requirements of our interface is to be a Single Page App, having it in mind we'll be using AngularJS to write the application.

This application will be consuming our API but not directly, once we're building an OAuth2 protected API the Single Page App could not consume the services without expose the oauth client credentials to the world, to keep it safe we'll need to use a proxy between the Single Page App and the backend API.

### Single Page App Flow



## Frontend and backend flow