Spring Taxi Dispatch Simulator

This was the Intern project finished in Summer 2018 regarding a web service for the real-time taxi dispatch simulator. Many thanks to my supervisor and other senior engineers who gave me help with all efforts they got. Time goes fast all the time; wish them all good luck!

(For confidential issues, the data used and presented are fake data generated randomly, but it does not effect the generosity of the project)

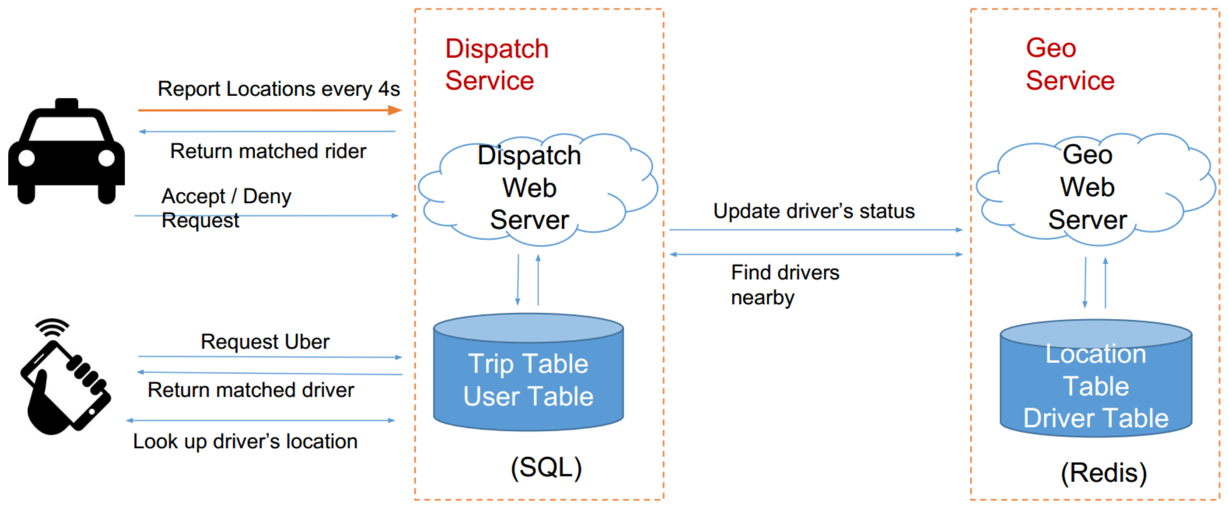
1 Introduction

Basically, the web application fulfills the dispatching simulating, order generation, driver monitoring, trajectory tacking, dashboard showcase, and etc.

For technology stacks, I developed the system with Java, Spring Boot/Data/Cloud, MongoDB, MySQL, RabbitMQ, and Netflix OSS with decoupled backend micro-service. In the meanwhile, a taxi dispatch model was proposed for solving the NP-hard combinatorial matching problem by GBDT and Hill Climbing method. With the help of implemented REST APIs for user and trip info in MySQL as well as trajectory data stored in MongoDB, we could visualized the real-time trajectories with SockJs, Stomp.js, Leaflet.js, VisualSearch.js, and Google Map API. Last but not least, the system is maintained by Netflix OSS: Incorporated Eureka as service registration and discovery; Used Hystrix as circuit breaker; Realized Zuul services gateway; Applied Spring Boot Actuator to monitor application health.

For system design, there are two key part that are upper level services, which are Dispatch Service and Geo Service.

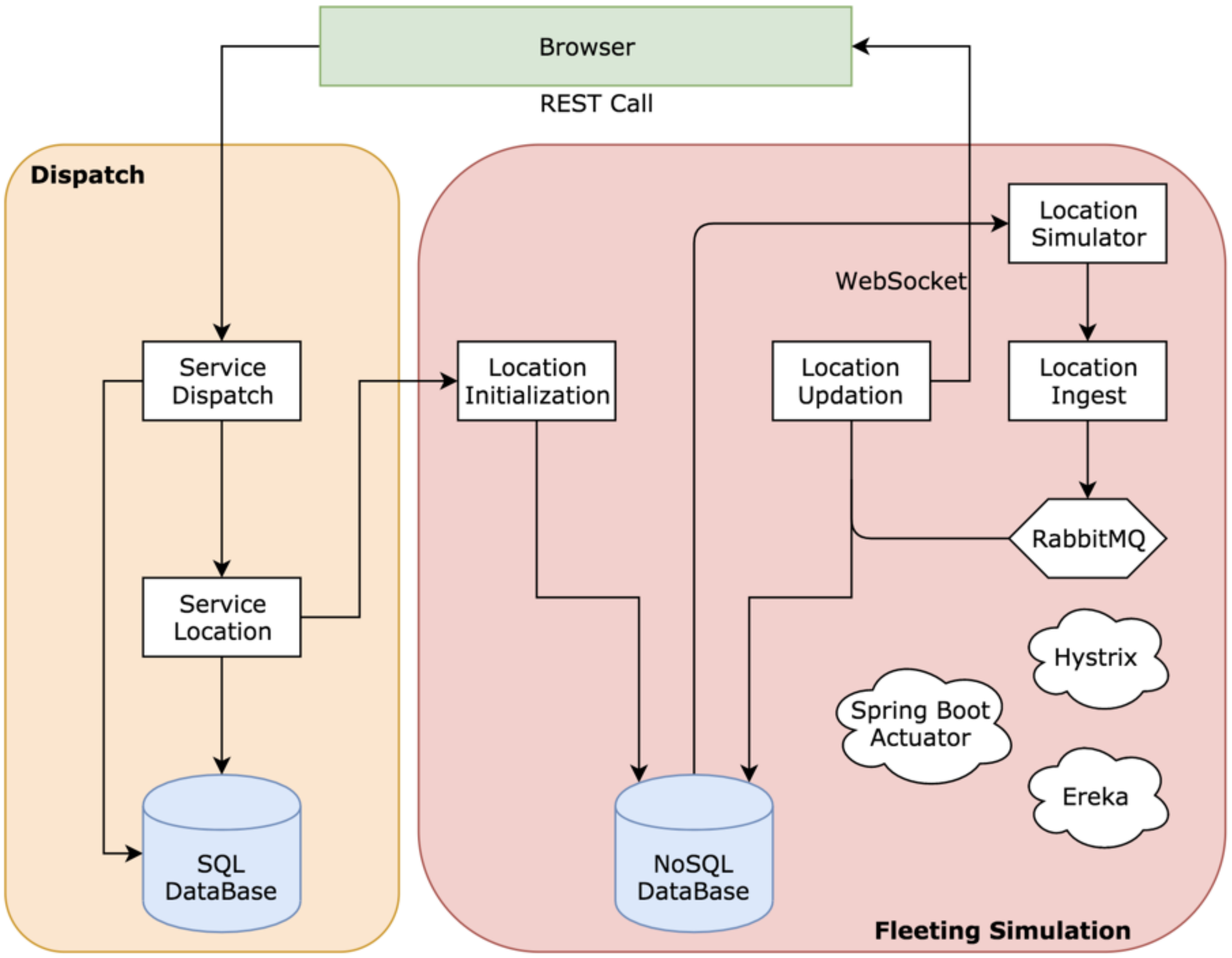
- Drivers send the location info to dispatch service, returning with paired user info
- User send request to dispatch service, returning with request



Location table will be writing heavily while the Trip table will be reading heavily (Drivers request queries)

The steps in the logic behind:

1. User send the request, and server will create a trip
    i. Return the Trip ID to user
    ii. User ask the server whether it is successful for pair
2. Server finds the pair, writing to trip. Trip state is waiting for response. Modify the driver table being "busy" with trip ID
3. Driver reporting location
    i. Find the trip ID for this driver in driver table
    ii. Find the trip in Trip table and return to the driver
4. Drivers accept the request
    i. Modify the Driver Table with the trip infomation
    ii. User finds he/she has been paired with the given driver
5. Driver deny the request
    i. Modify the Driver table with the trip info, marked by the info driver denying the trip
    ii. Re-pair with step repeated



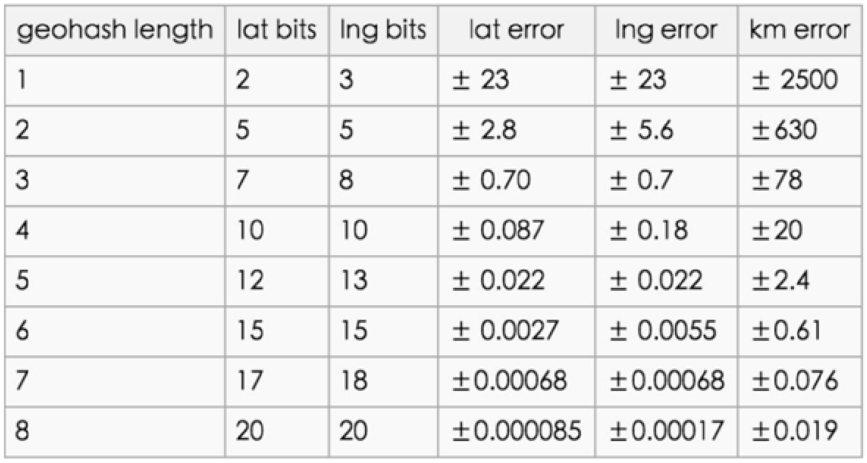
2 Algorithm Model - Dispatch

2.1 Classical Dispatching

In short, using GeoHash to find the nearest among certain range. we can apply a *Level-varying* store for driver table.

(lat,lng) → geohash → [driver1, driver2, ...]

driver1 → (lat, lng)



geohash length	lat bits	lng bits	lat error	lng error	km error
1	2	3	± 23	± 23	± 2500
2	5	5	± 2.8	± 5.6	±630
3	7	8	± 0.70	± 0.7	±78
4	10	10	± 0.087	± 0.18	±20
5	12	13	± 0.022	± 0.022	±2.4
6	15	15	± 0.0027	± 0.0055	±0.61
7	17	18	±0.00068	±0.00068	±0.076
8	20	20	±0.000085	±0.00017	±0.019

Example:

Google HQ: **9q9hv**u7wbq2s

We want to find all the locations with GeoHash beginning with "9q9hv"

SQL: it can be chosen, however, it is slow for *LIKE* query and it is not suitable for constant updated

NoSQL - Redis

- If Driver's location is 9q9hvt, then we store in 9q9hvt, 9q9hv, 9q9h keys, for this three, when finding, we can level by level to find
- 6 bit geohash is within 1km, enough for taxi
- 4 bit geohash goes beyond 20km, we will not go far for the reason that we will not ask for taxi too far away
- key = 9q9hvt, value = set of drivers in this location

Cassandra: it is also slow, driver table is constantly updated and index key in Cassandra is not good for modified so often

Memcached: there is no persistence storage (No P) and does not support Set (it is troublesome)

Mongo: It can be choosen

2.2 Optimal Dispatching

The quality of order dispatching can directly influence the user experience of riders as well as taxi operating efficiency. thus, how to dispatch orders efficiently is a central task. Some previous work on order dispatching focused on how to find a nearest driver or a shortest-travel-time driver for each individual order. When an order comes in, such a system chooses one of the nearest drivers, without judging whether these drivers were more suitable for other orders, therefore these methods cannot guarantee the global shortest-travel-time for all orders. The authors in proposed a novel model, based on a multi-agent architecture called NTuCab. In order to minimize the waiting-time or the pick-up distance globally, this model considers each agent as a computation unit. Each computation unit processes N order/driver pairs and each order is dispatched to only one driver. An order will be dispatched to another driver if the matched driver does not accept it.

A drawback of the methods mentioned above is the long dispatch time and low success rate, because the methods do not optimize the total success rate. At real situation, millions of drivers provide transportation services for over ten million passengers every day. In rush hours, we needs to match over a hundred thousand passengers to drivers every second. therefore the total success rate of these orders becomes the main metric to evaluate the performance of the underlying order dispatch system.

So we propose a novel combinatorial optimization model to solve the order dispatch problem. In this model, we dispatch one order to several drivers with the goal of maximizing the total success rate of these orders. When multiple drivers receive the same order, the first one to accept gets the order. If an order is not accepted, it enters the next round of dispatching until it is accepted or canceled.

Notations

We first introduce some notations. The goal of our order dispatch system is to maximize the success rate, denoted as ESR. If there are N orders to be dispatched to M drivers, we represent the dispatch result as a matrix.

An order is dispatched to a number of drivers, and each driver decides whether or not to accept it according to his or her own preference. For each order, whether it is accepted by one of the drivers is directly related to each driver's probability of acceptance. Thus, the key problem for order dispatching is to estimate the probability of each driver's acceptance of an order. If we can estimate the matrix with its elements indicating the probability of each driver accepting each order, then we can estimate the probability of an order to be accepted by one of the drivers therefore, we divide the order dispatch model into two sub-models. One model predicts each driver's action, in which we estimate the probability of a driver accepting an order. Another model formulates an optimization problem for maximizing the target ESR using the estimated acceptance probabilities, and then solves the underlying optimization problem.

Accept Rate

In short, we apply Machine Learning methods, linear logistic regression (LR) and gradient boosted decision tree (GBDT) to predict the accept rate from the the features. Actually, LR will be better.

$$p_{ij} = p(y = 1|o_i, d_j) = \frac{1}{\exp(-\mathbf{w}^{\mathbf{T}}\mathbf{x}_{ij})}$$

The prediction model considers various factors, which can be summarized as follows:

- Order-Driver related features: the pick-up distance, the broadcasting counts of the order to the driver, whether the order is in front of or behind the driver's current driving direction.
- Order related features: the distance and the estimated time arrival (ETA) between the origin and the destination, the destination category (airport, hospital, school, business district, etc.), tra.c situation in the route, historical order frequency at the destination.
- Driver related features: Long-term behaviors (include historical acceptance rate of a driver, active locations of a driver, preference of di.erent broadcast distances of a driver, etc.) and short-term interests of a driver such as orders recently accepted or not, etc.
- Supplemental features, such as day of the week, hour of the day, number of drivers and orders nearby.

Optimization Model

In our system, one order can be dispatched to several drivers, thus all these drivers contribute to the probability of order acceptance.

Directly, I give the formulas for our optimization model.

$$\begin{cases} \max\limits_{a_{ij}} E_{SR} = \dfrac{\sum_{i=1}^{N}[1 - \prod_{j=1}^{M}(1 - p_{ij})^{a_{ij}}]}{N}, \\ \text{s.t. } \forall j, \ \sum\limits_{i=1}^{N} a_{ij} \le 1, \ a_{ij} \in \{0, 1\}. \end{cases}$$

Many combinatorial optimization problems are NP hard, and there is no e.cient general algorithm to solve this class of problems in polynomial time. A typical approach is to use a heuristic algorithm to .nd an approximate solution. Commonly used methods include hill-climbing methods, genetic algorithms, simulated annealing algorithms, etc. By balancing the accuracy and the performance, we choose a hill-climbing method to solve the problem.

Codes

The dispatching related codes are packed in these two files:

- Service-Location-Service
- Service-Dispatch-Model

3 Fleeting Simulation - Micro-Service

This part gives a overview of the fleeting, which is about the geoloaction. Generally, it consists of the data persistence, data ingest, data simulating, data updating, and so on.

This part will also deliver the UI with simulation of the drivers' moving when knowing the initial pick-up location and the destination of the user after the order is generated.

To fulfill the function, we utilized Spring Boot, Spring Cloud, Spring Data, Netflix OSS, and etc to frame our system by the micro-service. The micro-service listed are:

- Fleeting-Location-Initialization
- Fleeting-Location-Simulator
- Fleeting-Location-Ingest
- Fleeting-Location-Updater
- Dashboard
- Platform

Each of the backend services is corresponding to each business function

Design principle

- Breakdown system requirements into Microservices
- Each Microservice is single responsibility
- Each Microservice can evolve in its own pace
- Microservices can only be communicated through REST API or message queue
- Cloud infrastructure services should be least intrusive to each business service

Each Microservice is single responsibility

- Do one thing and one thing only
- Less complex business logic leads to scalability
- Highly cohesive internally. Only expose necessary APIs to other backend services
- Loosely coupled with other services

Microservices can only be communicated through REST API or message queue

- No functional / method calls
- Standard protocol
- Message is more stable than method calls
- Decouple different services
- Async communication

Cloud infrastructure services should be least intrusive to each business service

- Maintainability
- Clean code
- Developers focus on implementing business logic

3.1 Fleeting-Location-Initialization

Data

The fake data generated from the dispatch service can be passed by .json file. Here we can use a fake data to do the job for showing. A example of fake data as showed in fleet.json

```
{
  "vin" : "7c08973d-bed4-4cbd-9c28-9282a02a6032",
  "latitude" : "38.9093216",
  "longitude" : "-77.0036435",
  "heading" : "E",
  "gpsSpeed" : "0",
  "gpsStatus" : "OK",
  "odometer" : "76056",
  "totalEngineTime" : "2139.25",
  "totalIdleTime" : "409",
  "totalFuelUsage" : "9798.667824",
  "address" : "270 New York Ave NE, Washington, DC, 20002, USA",
  "timestamp" : "2015-07-02T15:49:35Z",
  "tspProvider" : "cyntrx",
  "vehicleMovementType" : "STOPPED",
  "serviceType" : "ServiceInfo",
  "unitInfo" : {
    "unitVin" : "7c08973d-bed4-4cbd-9c28-9282a02a6032",
    "engineMake" : "DET",
    "customerName" : "Koss and Sons",
    "unitNumber" : "22832911"
  },
  "unitFault" : {
    "vin" : "7c08973d-bed4-4cbd-9c28-9282a02a6032",
    "spn" : "524287",
    "fmi" : "31"
  },
  "faultCode" : null
}
```

Or you can use the data from file *SHData*, but the other service does not support the schema for that. A sample of the driversInit.json

```
{
    "belong": "inner",
    "carInfo": {
      "color": "grey",
      "gpsStatus": "N/A",
      "did": "5b62a1fcebc64a8235626b9b"
    },
    "commutePos": {
      "latitude": "31.162136",
      "longitude": "121.463789",
      "did": "5b62a1fcebc64a8235626b9b"
    },
    "did": "5b62a1fcebc64a8235626b9b",
    "latitude": 31.194255,
    "longitude": 121.515452,
    "oid": null,
    "serviceType": null,
    "updated_time": null,
    "vehicleMovementType": null
  }
```

API

You will be directed into HAL-Browser, which will guide you through the APIs to interact with database

Basically, we only care about the geo location, which is latitude and longitude. The queries can be made like this

```
@RepositoryRestResource(collectionResourceRel = "locations")
public interface ServiceLocationRepository
extends PagingAndSortingRepository<ServiceLocation, Long> {

    @RestResource(rel = "by-location", description = @Description("Find by location, comma separated, e.g.
    ServiceLocation findFirstByLocationNear(@Param("location") Point location);

}
```

Or

```
@RequestMapping(value="/bulk/serviceLocations", method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED)
public void upload(@RequestBody List<ServiceLocation> locations) {
    this.repository.save(locations);
}

@RequestMapping(value="/purge", method=RequestMethod.POST)
public void purge() {
    this.repository.deleteAll();
}
```

Application Settings

The service will run on `localhost: 9000`

The `application.yml` file

```
server:
  port: 9000
spring:
  application:
    name: fleet-location-service
```

Posting Data

Another simple data for demo is in `serviceLocation.json`

```
{
    "latitude": 38.907774,
    "longitude": -77.023736,
    "address1": "1317 9th St NW",
    "city": "Washington",
    "state": "DC",
    "zip": "20001",
    "type": "Service"
  }
```

Using Postman or go through command line with code below will inject the data in

```
curl -H "Content-Type: application/json" localhost:9001/bulk/serviceLocations -d @serviceLocations.json
```

3.2 Fleeting-Location-Simulator

Google Map API

Simulator replies heavily on the Google Map API.

Firstly, we denote the dependencies brought from the Google Map API

```
<dependency>
    <groupId>de.micromata.jak</groupId>
    <artifactId>JavaAPIforKml</artifactId>
    <version>2.2.1</version>
</dependency>
<dependency>
    <groupId>com.google.maps</groupId>
    <artifactId>google-maps-services</artifactId>
    <version>0.1.7</version>
</dependency>
<dependency>
    <groupId>com.spatial4j</groupId>
    <artifactId>spatial4j</artifactId>
    <version>0.4.1</version>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>3.3.5</version>
</dependency>
<dependency>
    <groupId>net.sf.sprockets</groupId>
    <artifactId>sprockets</artifactId>
    <version>2.6.0</version>
</dependency>
```

Using the Direction, we can demo the route and movement of the driver, when we get the start point as address and end point also as address. The `direction.json` shows the user's pick-up location and destination. Like

```
{
    "from": "200 Mac Dill Blvd SE, Washington, DC 20340",
    "to": "901 Massachusetts Ave NW, Washington, DC 20001, USA"
}
```

Google Map API use the term "legs" to form a polyline, and polyline can be parsed into a String for use.

> Each element in legs array specifies a single leg of the journey from the origin to the destination in the calculated route. For routes that contain no waypoints, the route will consist of a single "leg," but for routes that define one or more waypoints, the route will consist of one or more legs, corresponding to the specific legs of the journey.

More on: https://developers.google.com/maps/documentation/directions/intro#Legs

In short, Google Map API will help us generated the routes and the routes will be abstracted into a String. Then, we can do the simulation, based on it. A example is below:

```
"gpsSimulatorRequests": [
    {
      "vin": "7c08973d-bed4-4cbd-9c28-9282a02a6032",
      "speedInKph": 50,
      "move": true,
      "exportPositionsToKml": true,
      "exportPositionsToMessaging": true,
      "reportInterval": 1000,
      "secondsToError": 0,
      "vehicleStatus": "SERVICE_SOON",
      "polyline": "}_ulFvq|tM@wLtOCnG@RHhArAzDvEZZTN|DjAz@Tv@Xp@f@j@j@b@j@vAhBhB~Bh@x@Zp@b@jAdCfIjDlLrClJV
      "faultCode": {
        "engineMake": "DET",
        "faultCode": "FMW",
        "faultCodeId": "DET",
        "faultCodeClassification": "ServiceInfo",
        "description": "Firmware Upgrade Required",
        "repairInstructions": "Verify Software update has been completed if available for this engine.",
        "fmi": "14",
        "sa": null,
        "spn": "171"
      }
    }
```

Structures

Due to the fact we use the Google Map API, we should follow the rules made by the framework, then we generated a series of classes to support our simulation. The table below showcases some packages and main function.

Package	Function
model	Classes created to follow the use of Google Map API
support	Navigation utils by Google Map
rest	REST APIs
service	Simulator/Path/Position Implementation
task	Muti-thread vehicle moving

API

Basically, we have the 4 external APIs for routings

- Start Simulation - it will start the simulation and sending messages
- Cancel Simulation - the opposite of start simulation
- Open UI - go to UI dashboard
- Status - return json object of the status

Application Settings

This system will run on `localhost: 9005`

We leave googleApiKey in this file as well

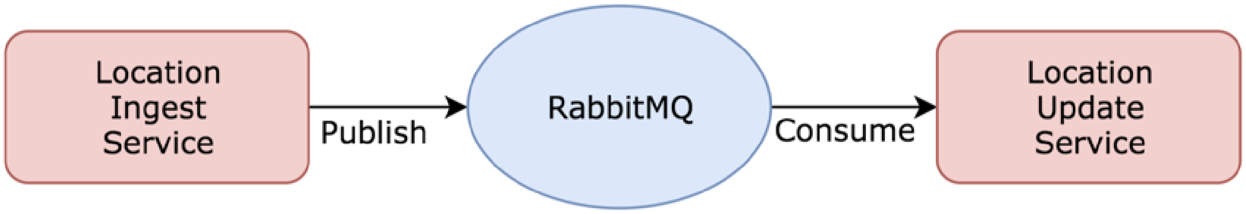
```
server:
  port: 9005
spring:
  application:
    name: fleet-location-simulator
gpsSimmulator:
  googleApiKey: AIzaSyDvdz2cb4zfp8NQVFu4kdzXFoIcHG744I4

com:
  jiahui:
    fleet:
      location:
        ingest: http://localhost:9006
---
spring:
  profiles: test
ribbon:
  eureka:
    enabled: false
fleet-location-ingest:
  ribbon:
    listOfServers: localhost:9006
```

it will interact with fleeting-location-ingest and fleeting-location-updater, we will leave this to the next sections

3.3 Fleeting-Location-Ingest

Overview

Publish current locations to locations queue in RabbitMQ



Rabbit MQ

From now on, the famous message broker will come to be handy to help us go through the message deliver.

Ingest service acts between the simulator and updater. We need to specify the source and sink when it comes to interaction between them. The binding name is: `vehicle`

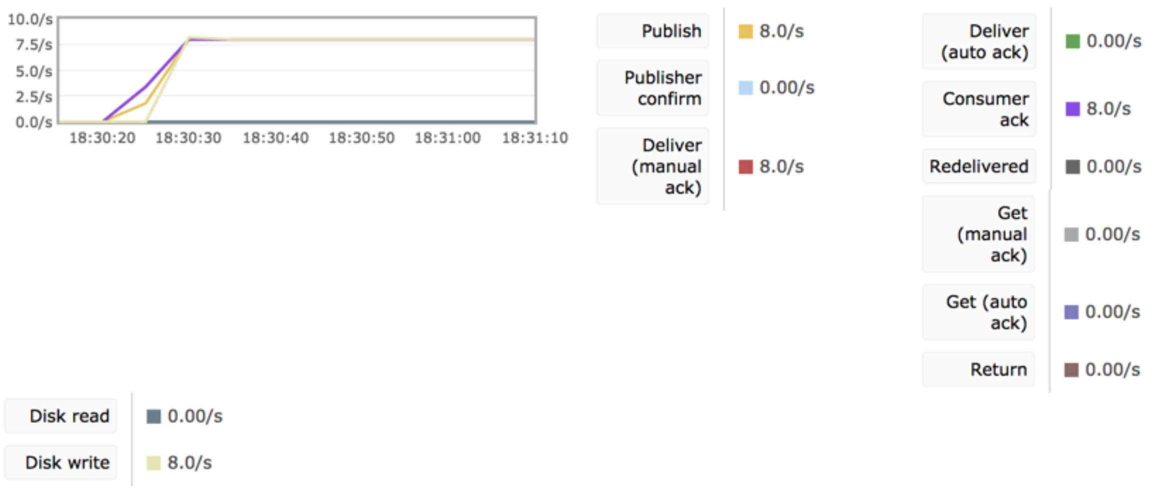
Spring Cloud provides templet to fast bind the publisher and consumer

```
@EnableBinding(Source.class)
```

```
@Autowired
private MessageChannel output;
```

The dashboard can show some summaries for the messages

```
http://localhost:15672/
User/password:(guest/guest)
```



Publish	🟨 8.0/s
Publisher confirm	🟦 0.00/s
Deliver (manual ack)	🟥 8.0/s

Deliver (auto ack)	🟩 0.00/s
Consumer ack	🟪 8.0/s
Redelivered	⬛ 0.00/s
Get (manual ack)	⬜ 0.00/s
Get (auto ack)	🟦 0.00/s
Return	🟫 0.00/s

Disk read	⬛ 0.00/s
Disk write	🟨 8.0/s

API

The only thing this service does is to re-send the message of location, then we can send the position info through

```
@RequestMapping(path = "/api/locations", method = RequestMethod.POST)
public void locations(@RequestBody String positionInfo) {
    this.output.send(MessageBuilder.withPayload(positionInfo).build());
}
```

Application settings

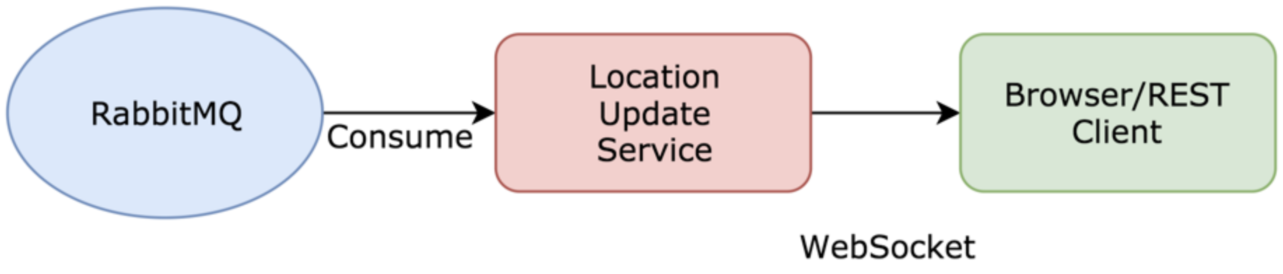
The system will run on `localhost:9006`

```
server:
  port: 9006
spring:
  application:
    name: fleet-location-ingest
  cloud:
    stream:
      bindings:
        output: vehicles
```

3.4 Fleeting-Location-Updater

Overview

Consume current locations from locations queue in RabbitMQ and Setup Websocket connect and push locations to client



WebJars

Now we will move backend generally to frontend. WebJars can be useful for the frontend framework package management.
WebJars packages these common Web front-end resources into Java Jar packages, and then manages them with Maven tools to ensure that these web resource versions are unique and upgrades are easier.

WebSocket and Stomp.js

The key thing that matters is the Rest Api, which sends to `/queue/vehicles` .

```
@MessageMapping("/sendMessage")
@SendTo("/queue/vehicles")
public String sendMessage(String message) throws Exception {
    return message;
}
```

For WebSocket configuration, we need to register the message broker and register stomp end point.

Message Sink

Spring Cloud Stream is responsible for sending current position data to connected WebSocket client. We can denote the endpoint and sink with the annotation

```
@MessageEndpoint
@EnableBinding(Sink.class)
```

To link with the former service of ingesting

```
@ServiceActivator(inputChannel = Sink.INPUT)
public void updateLocationaddServiceLocations(String input) throws Exception {

    CurrentPosition payload = this.objectMapper.readValue(input, CurrentPosition.class);

    this.template.convertAndSend("/topic/vehicles", payload);
}
```

Updater

Spring Cloud will provide rest template for us

```
private RestTemplate restTemplate = new RestTemplate();
```

The *updateServiceLocations* will be used like this:

```
ResponseEntity<Resource<ServiceLocation>> result = this.restTemplate.exchange(
                        "http://service-location-service/serviceLocations/search/findFirstByLocationNear?lo
                        HttpMethod.GET, new HttpEntity<Void>((Void) null),
                        new ParameterizedTypeReference<Resource<ServiceLocation>>() {
                        }, currentPosition.getLocation().getLatitude(),
                        currentPosition.getLocation().getLongitude());
```

Current position will be enriched with the closest service location

Application Settings

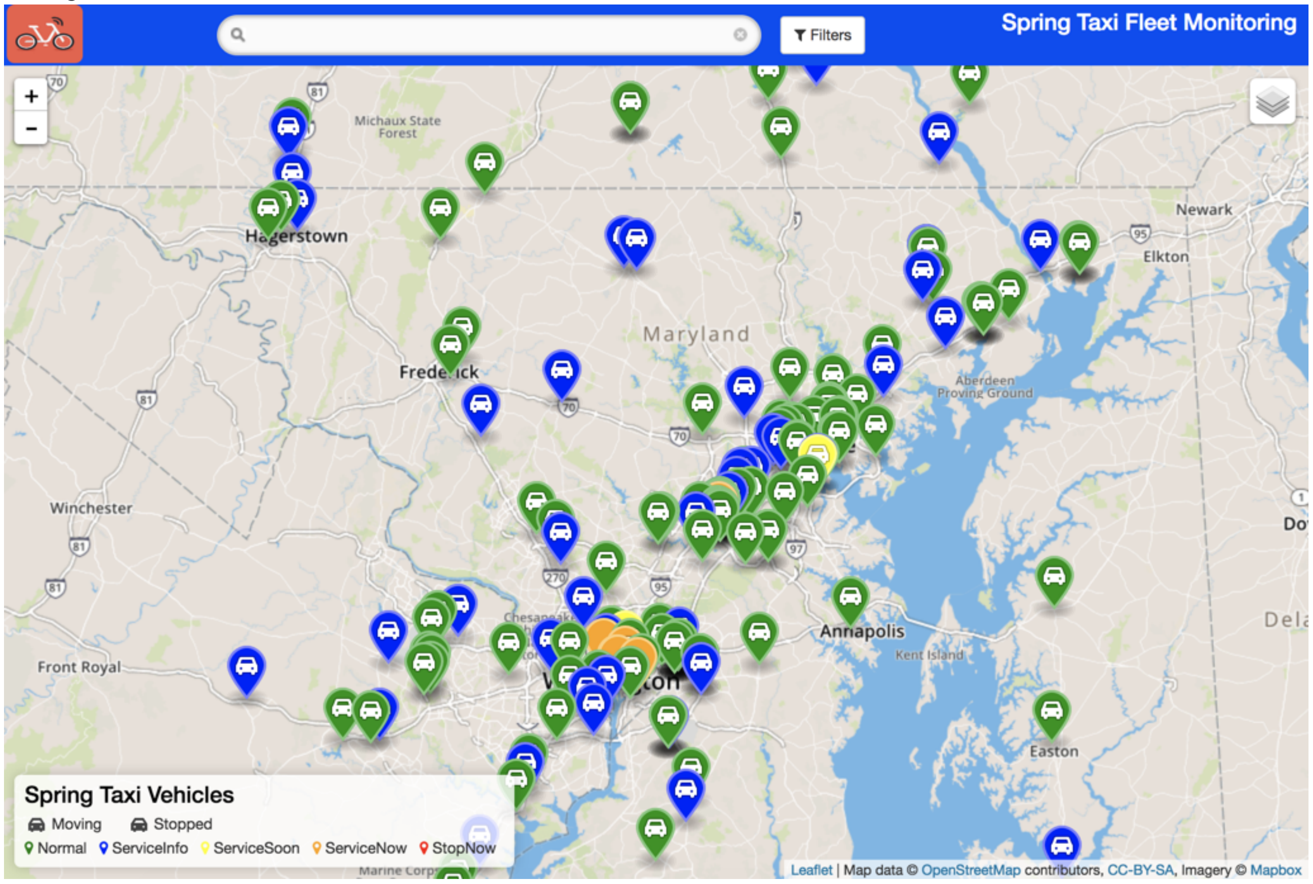
The system will run on `localhost: 9007`

```
server:
  port: 9007
spring:
  application:
    name: fleet-location-updater
  cloud:
    stream:
      bindings:
        input: vehicles
---
spring:
  profiles: test
```

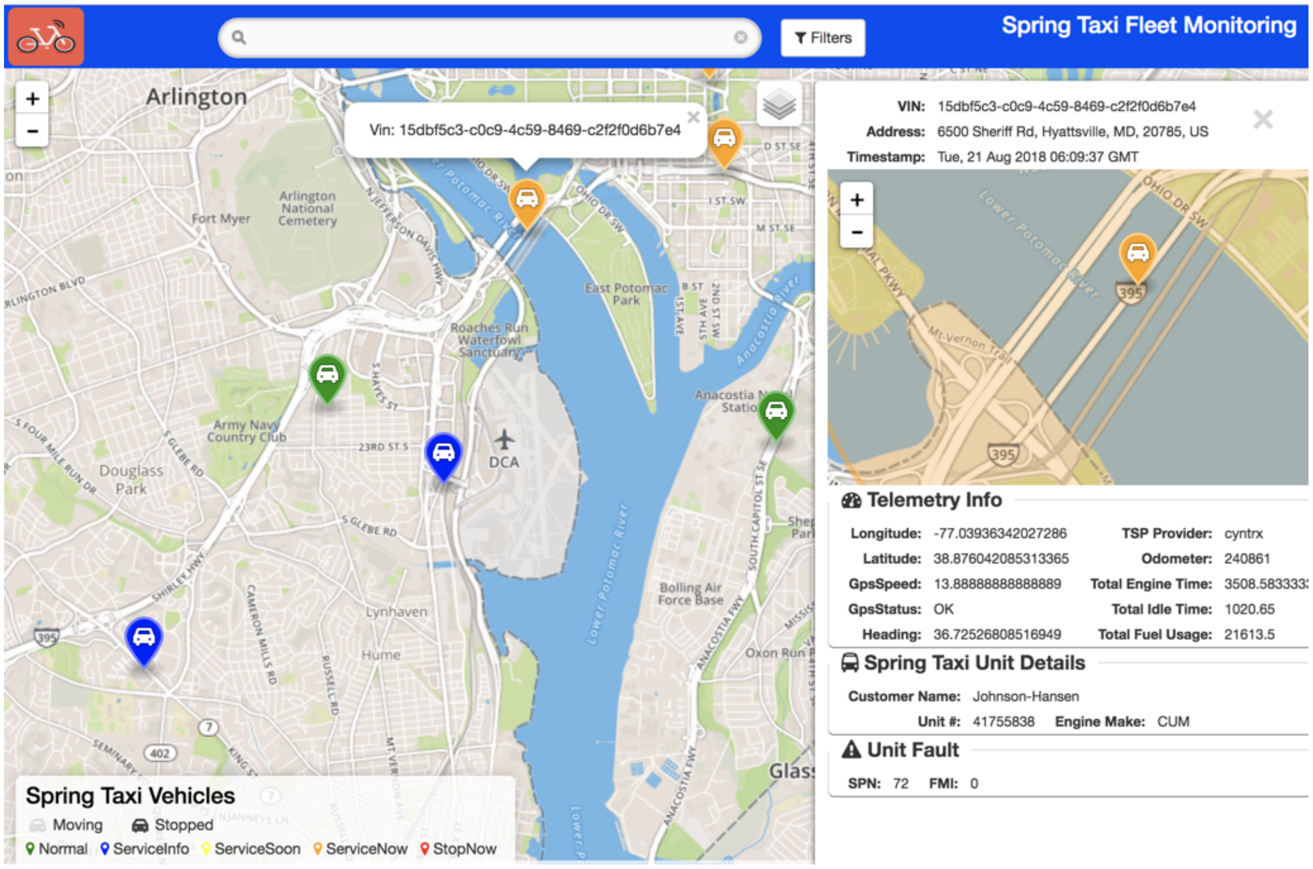
3.5 Dashboard

We use **Leaflet.js** framework as our map UI, the dashboard will show the moving our the cars and the details of each car as well

The figure below shows the overall distribution of the drivers


Filters
Spring Taxi Fleet Monitoring
Spring Taxi Vehicles
Moving
Stopped
Normal
ServiceInfo
ServiceSoon
ServiceNow
StopNow
Leaflet | Map data © OpenStreetMap contributors, CC-BY-SA, Imagery © Mapbox

We can target one driver to view the detail infomation



VIN: 15dbf5c3-c0c9-4c59-8469-c2f2f0d6b7e4
Address: 6500 Sheriff Rd, Hyattsville, MD, 20785, US
Timestamp: Tue, 21 Aug 2018 06:09:37 GMT

Telemetry Info

Longitude:	-77.03936342027286	**TSP Provider:**	cyntrx
Latitude:	38.876042085313365	**Odometer:**	240861
GpsSpeed:	13.88888888888889	**Total Engine Time:**	3508.583333
GpsStatus:	OK	**Total Idle Time:**	1020.65
Heading:	36.72526808516949	**Total Fuel Usage:**	21613.5

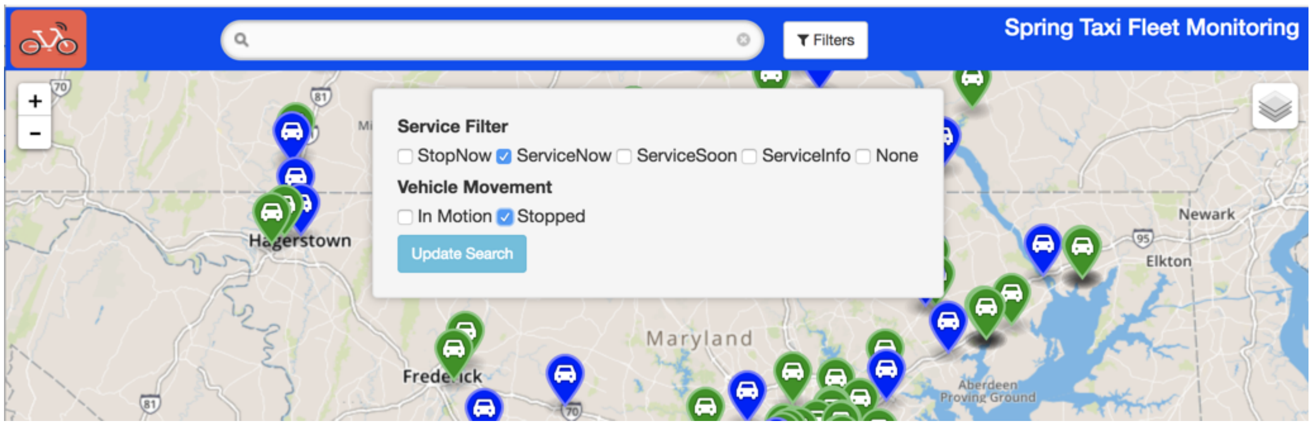
Spring Taxi Unit Details

Customer Name: Johnson-Hansen
Unit #: 41755838 **Engine Make:** CUM

Unit Fault

SPN: 72 **FMI:** 0

There are also some basic search features for filtering



Service Filter
☐ StopNow ☑ ServiceNow ☐ ServiceSoon ☐ ServiceInfo ☐ None
Vehicle Movement
☐ In Motion ☑ Stopped
Update Search

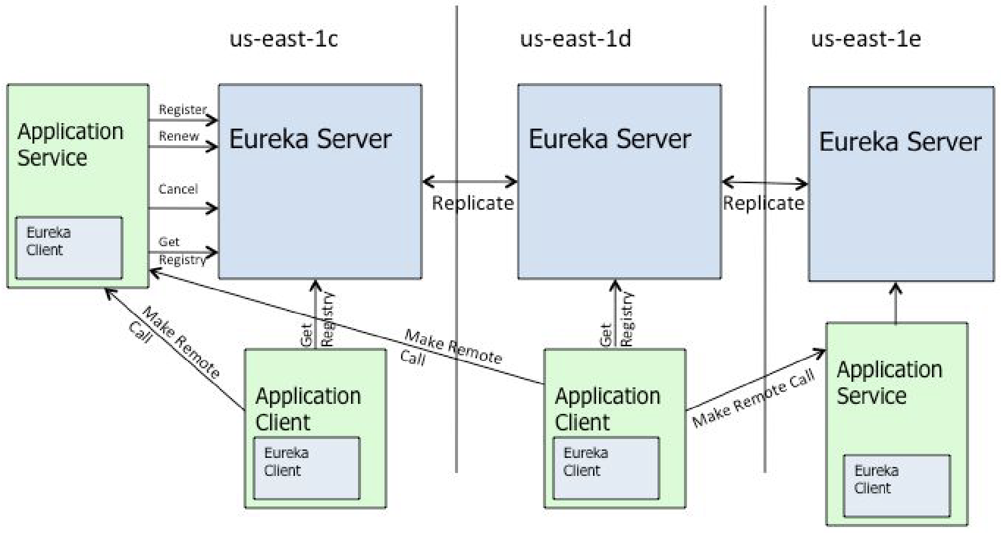
3.6 Platform

Docker

Dockerized the system by this docker compose file. The docker will pull the image and start RabbitMQ or MongoDB

```
rabbitmq:
  image: rabbitmq:3-management
  ports:
    - "5672:5672"
    - "15672:15672"
mongodb:
  image: mongo
  ports:
    - "27017:27017"
```

Eureka



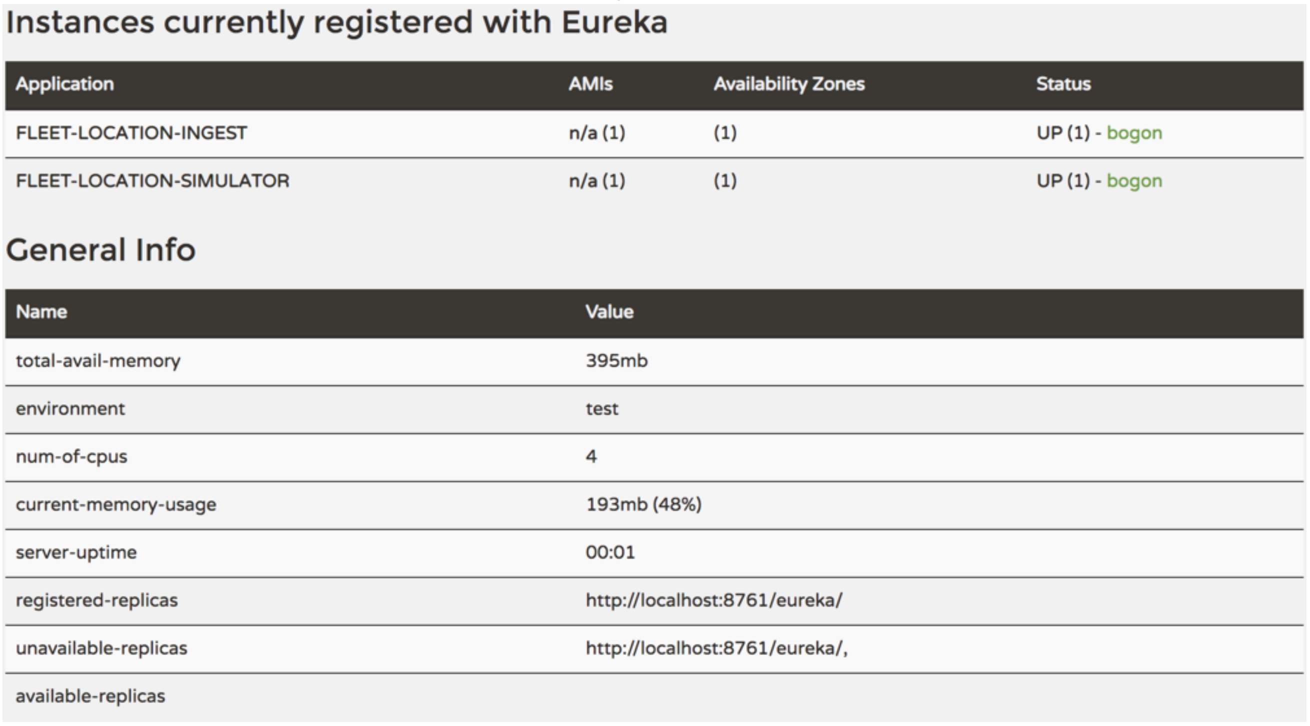
Eureka is used for service register and discovery, we need to run itself as a micro-service

```
@EnableEurekaServer
```

To let Eureka find your service, we simply need to add anotation in the main entry of your service

```
@EnableDiscoveryClient
```

we can see from `localhost:8761` , which will show the registered services



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
FLEET-LOCATION-INGEST	n/a (1)	(1)	UP (1) - bogon
FLEET-LOCATION-SIMULATOR	n/a (1)	(1)	UP (1) - bogon

General Info

Name	Value
total-avail-memory	395mb
environment	test
num-of-cpus	4
current-memory-usage	193mb (48%)
server-uptime	00:01
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/,
available-replicas	

Hystrix

Hystrix is for circuit breaker, it has its own dashboard to view. Likewise, we need to run itself as a micro-service

```
@EnableHystrixDashboard
```

To add it, we simply need to add anotation in the main entry of your service

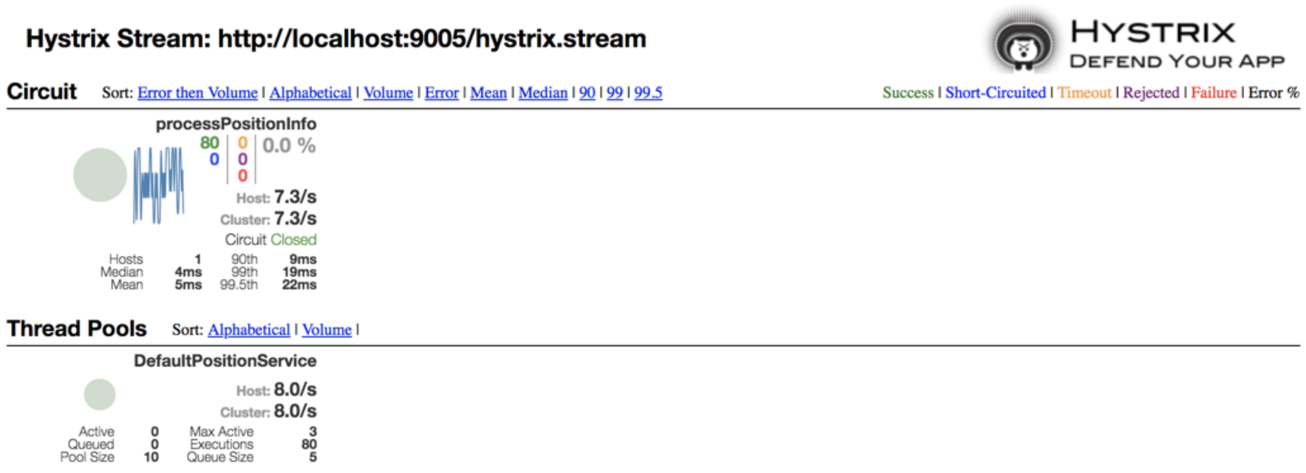
```
@EnableCircuitBreaker
```

An example usage in *updater* service

```
public void handleServiceLocationServiceFailure(CurrentPosition currentPosition) {
    LOGGER.error("Hystrix Fallback Method. Unable to retrieve service station info.");
}
```

The dashboard will run on `localhost:7979`

```
Hystrix Stream: http://localhost:9005/hystrix.stream
```



Hystrix Stream: http://localhost:9005/hystrix.stream

Circuit Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5

Success | Short-Circuited | Timeout | Rejected | Failure | Error %

```
processPositionInfo
80 | 0 | 0.0 %
 0 | 0
   | 0
Host: 7.3/s
Cluster: 7.3/s
Circuit Closed
 Hosts     1   90th    9ms
Median   4ms   99th   19ms
  Mean   5ms 99.5th   22ms
```

Thread Pools Sort: Alphabetical | Volume |

```
DefaultPositionService
Host: 8.0/s
Cluster: 8.0/s
   Active  0  Max Active  3
   Queued  0  Executions 80
Pool Size 10  Queue Size  5
```

Zuul

Zuul is a gateway layer. It is also a micro-service. Like other services, Service-1, Service-2, ... Service-N, it is registered on the eureka server. It can discover each other. Zuul can sense which services are online. At the same time, by configuring routing rules (examples are given later), the request can be automatically forwarded to the specified back-end micro-service for some common pre-processing (such as: rights authentication, token validity check, gray-scale verification part) Traffic guidance and the like can be handled in a so-called filter (ZuulFilter), so that the back-end service adds services later, and the zuul layer hardly needs to be modified.

Spring Boot Acuator

Spring Boot Actuator Endpoints

- /health
- /metrics
- /env
- /mappings
- /info

We can have a look at the health information for the service runs on `localhost:9005` by `localhost:9005/health`

```
{
    "description": "Spring Cloud Eureka Discovery Client",
    "status": "UP",
    "discoveryComposite": {
        "description": "Spring Cloud Eureka Discovery Client",
        "status": "UP",
        "discoveryClient": {
            "description": "Spring Cloud Eureka Discovery Client",
            "status": "UP",
            "services": [
                "fleet-location-simulator",
                "fleet-location-ingest"
            ]
        }
    },
    "diskSpace": {
        "status": "UP",
        "total": 121018208256,
        "free": 23863771136,
        "threshold": 10485760
    },
    "hystrix": {
    "status": "UP"
    }
}
```

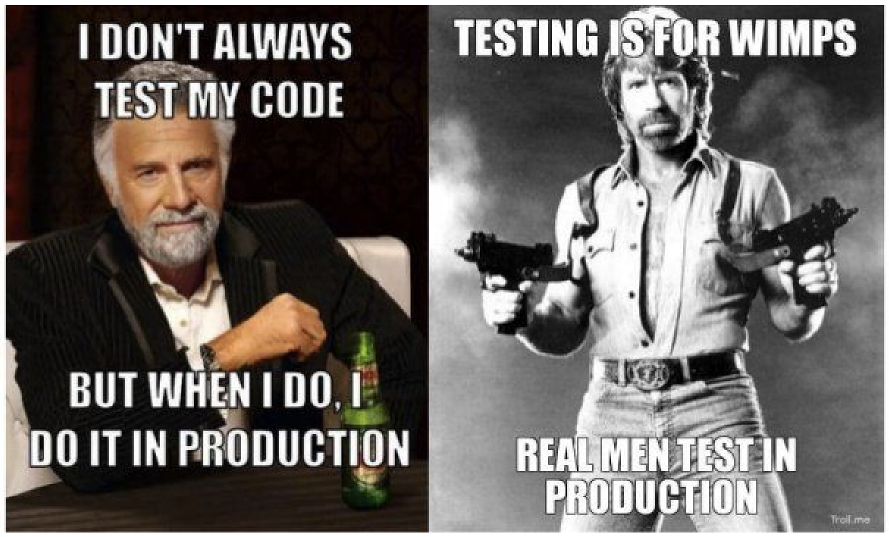
Write your custom health check:

```
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class MyHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        int errorCode = check(); // perform some specific health check
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }

}
```

More...

Continuous Delivery (CD)

Hahaha...


I DON'T ALWAYS TEST MY CODE
BUT WHEN I DO, I DO IT IN PRODUCTION
TESTING IS FOR WIMPS
REAL MEN TEST IN PRODUCTION
Troll.me

4 Discussion

4.1 Scalability

The default requirement is 150k QPS. The database writing/reading speed > 100k QPS, but if our Database is down.

So we may need **DB sharding**

- Split load
- Avoid single point failure

City Sharding

- Define a City
- Judge which City -- Whether a user is in a polygon
- City boundary -- nearby 2-3 cities/combination

POI Judgement

- Geo Fence
- Two Level - City/Airport Fence

DB down

- Replica by DB —— Master-Slave
- Replica by yourself
  - sharding key -> 123(city-id) to 123-0, 123-1, 123-2
  - read from any replica, if not this one, then switch
- Riak / Cassandra -- help with recovery

4.2 To Be Done

For the two big system: dispatch and fleeting simulation, we actually uploading pre-defined locations and status from JSON file, deserialize to JSON object, and store into database then getting locations based on ID, type, or all locations. Then, frontend can use running location service to initialize all locations during startup.

To meet the design requirement, we need to make the system connected and fulfill the design principles stated before.

On the other hand, front end page is lacking in features. We should add more functions in it with better framework to achieve routing and more advanced productive features.

For deployment AWS can be natively support, some setting-ups need to be done for the transfer.

```
eureka:
  client:
    #Region where eureka is deployed -For AWS specify one of the AWS regions, for other datacenters
specify a arbitrary string
    #indicating the region.This is normally specified as a -D option (eg) -Deureka.region=us-east-1
    region: default


    #For eureka clients running in eureka server, it needs to connect to servers in other zones
    preferSameZone: false

    #Change this if you want to use a DNS based lookup for determining other eureka servers. For example
    #of specifying the DNS entries, check the eureka-client-test.properties, eureka-client-
prod.properties
    #shouldUseDns: false

    us-east-1:
      availabilityZones: default

  instance:
    #Virtual host name by which the clients identifies this service
    virtualHostName: ${spring.application.name}
```

Appendix

```
##Service Start Sequence
1. docker-compose up
3. sh ./start-location-simulator.sh
4. sh ./start-location-ingest.sh
5. sh ./start-location-updater.sh
6. sh ./start-fleet-location-service.sh
7. go to fleet location service folder and run sh ./upload-fleet.sh
8. sh ./start-dashboard.sh

##UI
1. Open Dashboard UI on http://localhost:8080
2. Open Simulator UI on http://localhost:9005
3. Click run simulation
```