

# **A rule-based geospatial reasoning system for trip price calculations**



**Stefan Schenk**

Supervisor: Willem Brouwer

Advisor: Mewis Koeman

Department of Software Engineering  
Amsterdam University of Applied Sciences

This dissertation is submitted for the degree of  
*Bachelor Software Engineering*

April 2018



# Todo list

Refer to thresholds . . . . .	2
What are the main differences between postal systems used around the globe? . .	8
talk about use cases and usefulness of geospatial data . . . . .	9
See if other people solved locations . . . . .	10
Add ref to snippet . . . . .	11
Add ref to Geospatial Query Operators — MongoDB Manual 3.6 . . . . .	12
Add ref to image . . . . .	12
Add ref to snippet . . . . .	12
Add reference to Agarwal and Rajan . . . . .	13
Add reference to Geospatial Performance Improvements in MongoDB 3.2,” MongoDB	13
Add ref to Stephan Schmid Eszter Galicz . . . . .	13
Show diagram with hierarchy of companies and apps . . . . .	19
Write chapter about methods and technologies . . . . .	23
This is not researched yet, as it’s covered in later sprints . . . . .	29



# A rule-based geospatial reasoning system for trip price calculations

<i>Author</i>	<i>Stefan Schenk, 500600679, +31638329419</i>
<i>Place and date</i>	<i>Medemblik, 10 Mar 2018</i>
<i>Educational Institution</i>	<i>Amsterdam University of Applied Sciences</i>
<i>Department</i>	<i>HBO-ICT Software Engineering</i>
<i>Supervisor</i>	<i>Willem Brouwer</i>
<i>Company</i>	<i>taxiID, development team</i>
<i>Company address</i>	<i>Overleek 4</i> <i>1671 GD Medemblik</i> <i>Netherlands</i>
<i>Company Advisor</i>	<i>Mewis Koeman</i>
<i>Period</i>	<i>01 Feb 18 t/m 30 Jun 18</i>



## **Abstract**

A purely geometrical interpretation of user-defined locations would allow taxi-companies around the world to set up rules so that trip prices could be calculated without depending on distinct postal code systems. Geolocation datatypes provide part of the solution, but the benefits of geometrical definitions are lost when areas intersect. A hierarchy of precedence based rules tied to reusable locations would eliminate these competing rule matches.

A solution is proposed to implement a microservice with a single responsibility of calculating trip prices that is accessible to existing systems and portals in which users can define the pricing rules. The company for which this system is realized requires customers to be able to migrate to the new system without downtime, while keeping the existing rules that determine the prices of taxi trips.

The portals providing users access to company information must integrate a separate user interface allowing pricing rules to be managed. The microservice must be able to authenticate direct requests. The core system manages user and company data, complicating identity management in the microservice. A JSON Web Token would allow user identity to be stored in the payload of the token, thereby delegating authentication to the core system, safeguarding the single responsibility of the microservice.





# Table of contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context . . . . .	1
1.2	Problem Definition . . . . .	2
1.3	Assignment . . . . .	3
1.4	Research . . . . .	3
1.4.1	Questions . . . . .	4
1.5	Process . . . . .	5
<b>2</b>	<b>Encoding Locations</b>	<b>7</b>
2.1	Introduction . . . . .	7
2.2	A Brief History Of Geographic Locations . . . . .	7
2.3	Requisites of Location Types . . . . .	8
2.3.1	Location Related Scenarios . . . . .	9
2.4	Literature Review . . . . .	10
2.5	Database Prerequisites . . . . .	10
2.5.1	OpenGIS Compatible databases . . . . .	10
2.5.2	OpenGIS Incompatible databases . . . . .	12
2.6	Performance and Clustering Trade-offs . . . . .	13
<b>3</b>	<b>System Architecture</b>	<b>15</b>
3.1	Introduction . . . . .	15
3.2	Architectural Patterns . . . . .	15
3.2.1	Monoliths and Microservices . . . . .	16
3.3	Information Dependencies . . . . .	17
3.4	Authentication and Authorization . . . . .	18
3.4.1	JSON Web Tokens . . . . .	18
3.4.2	oAuth 2.0 . . . . .	19
3.4.3	API Gateway . . . . .	23

3.5	Suitability of Methods and Technologies . . . . .	23
<b>4</b>	<b>Trip Price Calculation System</b>	<b>25</b>
4.1	Introduction . . . . .	25
4.2	Breakdown . . . . .	25
4.3	Timeframes . . . . .	26
4.3.1	Conventional Approach . . . . .	26
4.3.2	Bitmap . . . . .	27
4.4	Data Model . . . . .	27
4.5	Logical Flow . . . . .	27
<b>5</b>	<b>Proposed Portal Solution</b>	<b>29</b>
5.1	Introduction . . . . .	29
5.2	Required Views . . . . .	29
5.3	Methods and Techniques . . . . .	29
5.4	Proposal Pricing Rules View . . . . .	29
5.5	. . . . .	29
<b>6</b>	<b>Realization</b>	<b>31</b>
6.1	Introduction . . . . .	31
6.2	Methods and Techniques . . . . .	31
6.3	Sprint 1 - Dynamic Price Calculations . . . . .	31
6.4	Sprint 2 - Authentication and Authorization . . . . .	32
6.5	Sprint 3 - Setting up the Portal . . . . .	32
6.6	Sprint 4 - Expanding the Portal . . . . .	32
6.7	Sprint 5 - Expanding the Portal . . . . .	33
6.8	Result . . . . .	33
<b>7</b>	<b>Conclusion</b>	<b>35</b>
<b>8</b>	<b>Recommendations</b>	<b>37</b>
	<b>References</b>	<b>39</b>
	<b>List of figures</b>	<b>41</b>
	<b>List of tables</b>	<b>43</b>
	<b>Appendix A Pregame</b>	<b>45</b>

Table of contents	vii
-------------------	-----

---

<b>Appendix B   Sprint Summaries</b>	<b>81</b>
--------------------------------------	-----------



# Chapter 1

## Introduction

What was once an ordinary startup known as Uber, is now the most famous taxi dispatch company in the world [1]. In the same year that Uber was founded, a similar startup in the Netherlands called taxiID was launched; an Amsterdam based company providing end-to-end cloud solutions for taxi companies. Hailing a taxi has rarely been performed by sticking out ones hand, hoping to catch the attention of a bypassing cab ever since. Recently, taxiID has started developing a new brand called YourDriverApp (YDA), a lighter and newer version of the original solution, being more focussed on smaller taxi companies. Despite the fact that YDA is new, it still depends on the price calculation functionality of the legacy system. This chapter expands on this matter and how it was translated into an assignment.

### 1.1 Context

taxiID was founded as a startup that successfully introduced smartphone taxi booking in The Netherlands, and offers a wide range of IT solutions to serve the taxi market, including a passenger app, a driver app, and administrative panels. More specifically: an app for passengers to order a taxi, an app for drivers to receive their job assignments, and services for all size businesses, offering convenient planning and dispatching without requiring local installations. Businesses that make use of taxiID's services can be found anywhere in the world. This introduces complicated challenges while developing applications that rely on clearly defined locations and infrastructures, often vastly differing between countries, if these countries have such a system to begin with. The taxiID development team responsible for solving these problems is located in Medemblik, consisting of two mobile app developers (iOS and Android), two backend developers, a designer and two project managers.

## 1.2 Problem Definition

YDA depends on the price calculation module that is part of the legacy system for which it was designed and implemented. When a passenger books a ride, the departure and destination locations that have been selected are sent to the legacy system. It then proceeds and constructs a list of prices for each vehicle type that is available, based on matching pricing rules that have been defined by the taxi company offering the rides. If directors of a taxi company using YDA want to modify their pricing rules, they will be obligated to use the taxiID portal, which has to store company information in a platform that is different from YDA. This makes little sense, as much as it is efficient from a technical point of view, and being easy to maintain and extend. The current price calculation module knows three types of pricing rules: fixed prices based on postal codes, tier prices based on kilometer thresholds, and dynamic calculations based on distance and duration of a ride. A company may have as many rules as required, only one rule will be used to calculate the final price, and the rules are matched in the same order respectively. The fixed rules are defined by downloading, modifying, and uploading a .csv file as presented in table 1.1, the other types of rules are simply managed through a web form.

Departure	Destination	Nr Passengers	Price	Vehicle Type
1462	1313	4	125	
1313	1462	4	125	
1462	1313	8	150	
1313	1462	8	150	
1462	1012	4	65	
1012	1462	4	65	
0	1462	4	65	
1462	0	4	65	
1462	AIR1	4	89	
AIR1	1462	4	89	

Fig. 1.1 Comma Separated File containing Fixed Prices in cents

When a passenger books a ride, the price calculation module will first compare the postal codes, amount of passengers, and vehicle types in the fixed pricing rules with the information provided by the passenger's application. The fixed price is returned as soon as a match is found. If no match is found in any of the fixed pricing rules, the system proceeds to calculate a price using a kilometer threshold based rule, given that at least one exists. This type of calculation decreases or increases the price per kilometer for every successive amount of kilometers that have surpassed a predetermined threshold. This concept will be discussed in chapter

**kks32: Refer to thresholds**

. If this rule does not exist, a dynamic rule is used to calculate the price based on distance and duration of the ride. Finally, on top of the prices that have been calculated, a discount may be applied. As a fixed amount, as a percentage of the price, or as a so called alternative fixed pricing table. When this last option is selected, the price will be calculated all over, using a newly referenced fixed pricing rule. This process is not just hard to understand for a user, who has to reason about the companies prices. But it is also hard to understand for programmers, who have to maintain the code that supports this functionality. A small mistake in the csv file could lead to great issues if the mistake goes through processing undetected.

## 1.3 Assignment

The title of this thesis reads:

*"A rule-based geospatial reasoning system for trip price calculations".*

A Trip Pricing System (TPS) must be designed and implemented to calculate trip prices based on user defined pricing rules. Concisely, YourDriverApp requires its own pricing calculation functionality that is similar to the existing taxiID implementation but must not be incorporated into a non-related monolithical, highly coupled system, as it is today. Also, the response body should have the exact same format, and the new system must be able to handle the exact same requests that are made to the current system. Clients must be able to set up pricing rules through the YDA portal, and potentially other portals as well. It is also important that the feature allowing clients to define locations, is usable in countries without a workable postal code system.

## 1.4 Research

Three main challenges that construct the assignment can be identified. Research must be conducted to attain the best possible way of mapping locations to pricing rules. What this means is that locations must be storable, comparable, and interpretable. The database must be able to store locations in an efficient manner, to which queries can be made as efficiently in order to find out whether a pricing rule applies to a given ride. For this to be the case, the stored locations must be comparable to the location of the passenger, or the destination. The user must be able to reason about his pricing rules, from which an understanding of his

defined locations logically follows. But edge cases must be covered completely. For example, a rule in the current system dictates that a user traveling to Schiphol should receive a discount. But how would the system detect that this is the case? Or what if hotel guests receive discounts, but the neighbour shouldn't be allowed to use these discounts? Secondly, a system has to be developed that encapsulates the solution that is the result of the finished research. It is helpful to extend the research of the problem beyond finding out how to incorporate the answers into a working system, where architecture has a major influence in the tools that are available. For example: if a solution to the main problem requires a database system capable of handling high quantities of geospatial queries, this requirement has to be satisfied in order to proceed in finding the final solution. Finally, a user interface has to be created that enables users to define the pricing rules. The complexity of the interface depends on how straight forward the price calculation system is constructed. The user interface should also be available in multiple portals. The best way of making the systems capabilities available to the user through the UI in the portal, must be investigated. The UI must be built keeping the user in mind, simplifying complex rule management as much as possible.

### 1.4.1 Questions

From the description of the problem, one main important research question can be derived:

*How can a generic location-based price calculation system be implemented  
that could be used in every country?*

This question encapsulates the four important challenges that have to be dealt with before the project can successfully be implemented. In order to give a clear direction to the research, sub-questions are separated into four groups; location mapping, architecture, trip pricing system, and user interface.

1. In what way can locations be represented to be universally interpretable and precise?
  - 1.1. Which location types matter for this project?
  - 1.2. What are the main differences between postal systems used around the globe?
  - 1.3. Can postal codes be abstracted to geospatial data while retaining the same usefulness in the system?



- 1.4. Which Database Management Systems (DBMS) cover the location storage use cases for this project?
2. What is most fitting solution to integrate the backend and frontend into the existing architecture?
  - 2.1. Which architectural patterns fit in with the existing architecture?
  - 2.2. Which data of adjacent systems are required to make TPS operational?
  - 2.3. How can authentication between services be implemented or improved?
  - 2.4. Which methods and technologies can be used to ensure suitability and improve maintainability and testability?
3. Which logic and information is required to calculate a trip price breakdown?
  - 3.1. How are rules matched?
  - 3.2. Which parts of the system have an impact on the calculation result?
  - 3.3. How should the concept of time schedules be realised?
4. How can complex pricing rules be communicated through the UI?
  - 4.1. Which views are essential?
  - 4.2. How should locations be defined and managed by the user?
  - 4.3. How should timeframes be represented in the interface?

The first group of questions is answered in the chapter Encoding Locations. The second and third are answered in the chapter Proposed Approach. At that point, enough knowledge is available to implement a solution.

## 1.5 Process

A written working method is provided to the product owner, see Appendix A, Pregame. The purpose of the documents is to clearly define the assignment and reflect on the interpretation of the assignment definition, so that miscommunications are found immediately. Requirements, scope and stakeholders of the project, as well as laying out the project timeline and estimated architecture based on use cases are clearly documented. Finally, a proposed solution is the result, which is agreed upon by the product owner before the backlog is created, containing a preview of the ongoing research. Reading the document is recommended if more knowledge about the process and context of the assignment is desired.



# Chapter 2

## Encoding Locations

### 2.1 Introduction

Encoding of locations has historically been of great importance, and is always being modernized. This chapter explains the general definition of locations, which types of locations are important for this project, and how to represent these locations so that they are universally interpretable.

### 2.2 A Brief History Of Geographic Locations

A location is roughly described as a place or position. Throughout history, various navigational techniques and tools like the sextant, nautical chart and mariner's compass were used, measuring the altitude of the North Star to determine the latitude  $\phi$ , in conjunction with a chronometer to determine the longitude  $\lambda$  of a location on the Earth's surface. The combination of coordinates is a distinct encoding of a location.

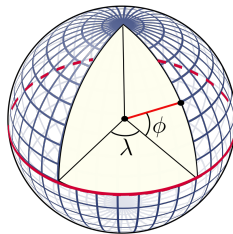


Fig. 2.1 A perspective view of the Earth showing how latitude and longitude are defined on a spherical model.

The history of this encoding goes way back to when it was first proposed in the 3rd century BC by Eratosthenes. He invented the discipline of geography, and was known

for also being the first person to calculate the circumference of the Earth with remarkable accuracy. Today, navigation relies on satellites that are capable of providing information to determine a location with a precision of 9 meters. Hybrid methods using cell towers, Wi-Fi Location Services, or the new Galileo global navigation satellite system, provide tracking with a precision down to the metre range. These locations are ordinarily communicated using the same established latitude and longitude encoding. For a human being, it is not practical to exchange day-to-day locations as geographical coordinates. For that, addresses much more suitable, but can be ambiguous, imprecise, inconsistent in format. Addresses commonly make use of Postal Code systems, which have reliably been assigned to geographical areas with the purpose of sorting mail. Although even today, there are countries that do not have a Postal Code system.

kks32: What are the main differences between postal systems used around the globe?

In contrast to the geographic coordinate system, postal codes describe streets and areas of varying shapes and sizes. A location being roughly described as a place or position, can be decomposed as an abstract term to describe physical or imaginary areas with varying radiusses and shapes. You could prepend 'the location of' to the following terms as an example: America, the birthplace of Sokrates, Wall Street, the center of the universe, the Laryngeal Nerve of the Giraffe, churches in the Netherlands. The final example presents the main challenge of this project, how to communicate the location of a collection with points or areas of differing shapes and sizes that may overlap?

## 2.3 Requisites of Location Types

While setting up a backlog for a project, a shared knowledge about the terminology used in the issues must be achieved in order to collaborate effectively. Words or symbols do not have an absolute meaning, and ambiguity of abstract linguistic terms should be elucidated. In Appendix A an agreement was made on what the terms "area" and "point" meant. A point is a unique place expressed as a distinct coordinate pair. An area is a set of many points that is tightly packed together. No degree of granularity is necessary, the coordinates are continuous. The point and shape can be translated to the addresses and postal codes in the legacy pricing system respectively if a radius was assigned to the centroid of the shape that is formed by the boundaries of the street, neighbourhood, province or country. There are some downsides to using addresses and postal codes:

1. Addresses can be ambiguous.
2. Addresses and postal codes can be imprecise.

3. Postal code systems are not uniform.
4. Some countries don't have postal code systems.

In contrast, spatial datatypes would provide unique and precise location definition that is uniform and universal. Many spatial database systems support a basic Geometry hierarchy of Points, Polygons, MultiPoint and MultiPolygon Classes, as described in the OGC [2] and ISO 19125 [3] standard, which are capable of mapping areas and points that are currently described by addresses and postal codes, and more. This answers the question whether postal codes can be abstracted to geospatial data, but has it retained its usefulness in the system?

kks32: talk about use cases and usefulness of geospatial data

### 2.3.1 Location Related Scenarios

The specific criteria to which the database geospatial functions must adhere are:

1. the system must distinguish points inside and outside of a location.
2. the system must detect whether a user travels from, or to, a point.
3. the system must be able to handle overlapping locations.
- 4.
5. users should be able to select predefined locations from external sources.

The set of all possible points on Earth:

$$P = \{(x,y) | -90 < x < 90, -180 < y < 180\}$$

$$A = \mathcal{P}(P)$$

Finally, collections of these possibilities are allowed to describe the problematic "all churches in the Netherlands" example:

$$C_p \subseteq P$$

and "all counties in which the majority voted Trump" example:

$$C_a \subseteq A.$$

This way a location could either be an area or a point, with which all possibilities are covered, except sets of these elements. As stated in Appendix A, the definition of an area is precise, unambiguous and easy to use in compare in computer programs. A single point may match another single point if it's the exact same point. A point may be sitting on top of a

line or is contained within an area. The only other option is the negation of these statements. Because use cases for lines will be non-existent, points and areas are the proper candidates for spatial queries.

A taxi company director wants to be able to set price or define discounts from or to a certain location. They would like to define prices based only on departure locations, or only on destination locations, or both. For example: 'to Schiphol, a trip should cost €10,-', or 'from van der Valk hotels, a trip should cost €5,-', or 'from van der Valk hotels to Schiphol, the km price should be €0,60'. In the current implementation, a record would be stored containing departure location, destination location and price for every combination, where locations were defined as zip codes. Instead, it would make sense to be able to reuse locations after they have been defined once.

## 2.4 Literature Review

What 3 words, a multi-award winning global addressing system, bases 3m x 3m squares, covering the planet, on a combination of three words.

Geospatial

Postal code

kks32: See if other people solved locations

## 2.5 Database Prerequisites

The database must be capable of determining whether a virtual perimeter contains a set of coordinates, more specifically, it must adhere to The Open Geospatial Consortium (OGC) Simple Feature Access ISO 19125-1 [2] and ISO 19125-2 [4], including spatial data types, analysis functions, measurements and predicates for this requirement. The scenario presented in image 2.2 should be replicable.

### 2.5.1 OpenGIS Compatible databases

MYSQL's innate integrity is a good reason to opt for a full MYSQL database setup. MariaDB is a fork of MYSQL that performs better according to benchmarks, however they don't always translate to real life situations. It's easy to migrate from MYSQL to MariaDB, so choosing MYSQL at first could be preferable as an instance of MYSQL is already used at TaxiID. PostgreSQL offers a spatial database extender for that is OpenGIS compliant called PostGIS that adds support for geographic objects and location queries.

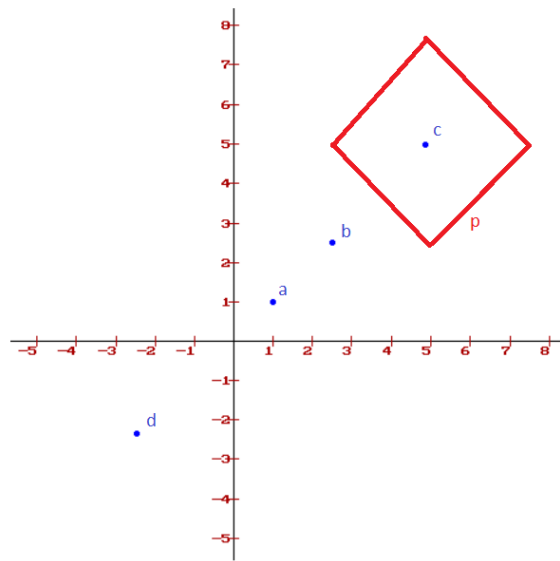


Fig. 2.2 Four Points, one Polygon p containing Point c.

All spatial data types inherit properties such as type and spatial reference identifier (SRID). For rigorous documentation, both PostGIS documentation [5] and MYSQL documentation [6] could be consulted. When a generic geometry column, or point column is created, points can be inserted as shown in snippet 2.1 and 2.2

<pre> 1  START TRANSACTION; 2  SET @a = ST_GeomFromText('POINT(1 1)'); 3  INSERT INTO point (point) VALUES (@a); 4  SET @b = ST_GeomFromText('POINT(2.5 2.5)')    ; 5  INSERT INTO point (point) VALUES (@b); 6  SET @c = ST_GeomFromText('POINT(5 5)'); 7  INSERT INTO point (point) VALUES (@c); 8  SET @d = ST_GeomFromText('POINT(-2.5 -2.5)')    ; 9  INSERT INTO point (point) VALUES (@a); 10 # also insert @b, @c, and @d 11 COMMIT; </pre>	<pre> 1  START TRANSACTION; 2  # First and last point must be the same 3  SET @a = PolygonFromText('POLYGON((2.5 5,5    7.5,7.5 5,5 2.5,2.5 5))'); 4  INSERT INTO polygon (polygon) VALUES (@a); 5  COMMIT; </pre>
---	--

Listing 2.2 Insert polygon

Listing 2.1 Insert four points

It is evident that c is contained in p. To determine which points are contained in p, the function as seen in Snippet

kks32: Add ref to snippet

can be used, which returns the point with coordinates [5,5] as expected.

```

1  // All points contained in polygon
2  SELECT ST_ASTEXT(POINT)
3  FROM POINT
4  WHERE
5  ST_CONTAINS(
6    (
7      SELECT POLYGON
8      FROM POLYGON
9      WHERE id = 1
10     ),
11    POINT
12   )

```

Listing 2.3 Select points contained in polygon

```

1  // All polygons containing point
2  SELECT ST_ASTEXT(POLYGON)
3  FROM POLYGON, POINT
4  WHERE
5    POINT.id = 3 AND ST_CONTAINS(
6      POLYGON.polygon,
7      POINT.point
8    )

```

Listing 2.4 Select polygons containing point

## 2.5.2 OpenGIS Incompatible databases

MongoDB doesn't offer OpenGIS implementations but has geospatial query operators that may provide enough functionalities for current requirements

kks32: Add ref to Geospatial Query Operators — MongoDB Manual 3.6

. The argument for choosing one over the other depends on the vast differences between SQL and NoSQL, next to performance and extensiveness of geospatial features. The setup displayed in image

kks32: Add ref to image

is recreated in MongoDB using queries shown in snippet

kks32: Add ref to snippet

.

```

1  db.point.insertMany([
2    { shape: { type: "Point", coordinates: [1, 1] } },
3    { shape: { type: "Point", coordinates: [2.5, 2.5] } },
4    { shape: { type: "Point", coordinates: [5, 5] } },
5    { shape: { type: "Point", coordinates: [-2.5, -2.5] } },
6  ])
7
8  db.polygon.insert({
9    shape: {
10     type: "Polygon",
11     coordinates: [ [ [2.5, 5], [5, 7.5], [7.5, 5], [5, 2.5], [2.5, 5] ] ]
12   }
13 })
14
15 db.point.createIndex({ 'shape': '2dsphere' })
16 db.polygon.createIndex({ 'shape': '2dsphere' })

```

Listing 2.5 Select points contained in polygon

```

1  // All points contained in polygon
2  var p = db.polygon.find({})

```



```

3                                     19  var p = db.point.findOne({ coordinates:
4  db.point.find({                    [5, 5] })
5  shape: {                           20
6  $geoWithin: {                      21  db.polygon.find({
7  $polygon: [                        22  shape: {
8  [2.5, 5],                          23  $geoIntersects: {
9  [5, 7.5],                          24  $geometry: {
10 [7.5, 5],                          25  type: "Point",
11 [5, 2.5],                          26  coordinates: [5, 5]
12 [2.5, 5]                          27  }
13 ]                                  28  }
14 }                                  29  }
15 }                                  30  })
16 })
17
18 // All polygons containing point

```

Listing 2.6 Select points contained in polygon

Next to database solutions for this requirement, services exist that are capable of geofencing. Although these services may not be free, and the added dependencies restrict extensibility.

## 2.6 Performance and Clustering Trade-offs

Agarwal and Rajan state that NoSQL take advantage of cheap memory and processing power, thereby handling the four V's of big data more effectively, but lack the robustness over SQL databases

kks32: Add reference to Agarwal and Rajan

. The report dives deeper into spatial queries and concludes that their tests suggest that MongoDB performs better by an average factor of 10, which increases exponentially as the data size increases, but lack many spatial functions that OpenGIS supports. Although improvements have been made

kks32: Add reference to "Geospatial Performance Improvements in MongoDB 3.2," MongoDB

after the cited paper Schmid et al. 2015

kks32: Add ref to Stephan Schmid Eszter Galicz

was published. The team argues that clustering is much easier in MongoDB, which may be important in the future when the company grows. As the required functionalities exist in both SQL and NoSQL, it is beneficial to opt for MongoDB for its performance and alignment with the teams experience. Although if robustness is desired, or extra GIS functionalities required, SQL should be taken into consideration.



# Chapter 3

## System Architecture

### 3.1 Introduction

In order to integrate TPS in the existing architecture, an analysis must be made so that the most suitable patterns are applied. The term TPS refers to the backend implementation that calculates the prices, and the frontend in which the rules can be defined. There exists an inverse relationship between the amount of adjacent systems from which information is required and integrability. This required information impacts the way TPS is implemented. When TPS requires user information from one system, vehicle information from another, and company information from another still, a gigantic monolithical system would simplify the task of sharing the data with its components. Authentication wouldn't be required because no network connection is required to communicate the information. In contrast, separate services offer more independence and modularity.

### 3.2 Architectural Patterns

The current system architecture consists of three API's and nine services that connect to four databases, as can be seen in figure 3.1. They provide functionalities to portals and mobile apps. The user interface, business logic, and data storage are separated, following the three-tier or multi-tier architecture [7]. The bigger and smaller shapes in the figure represent monolithical API's and services respectively. The orange colored services are used internally, the green ones can be used by other companies. The smaller services adhere to the pattern that is called service-oriented architecture (SOA), where application components provide services over a network typically. As discussed in the introduction of this chapter, it would

have been easier to integrate these services right into the monoliths that depend on them, eliminating the need for network requests and authentication.

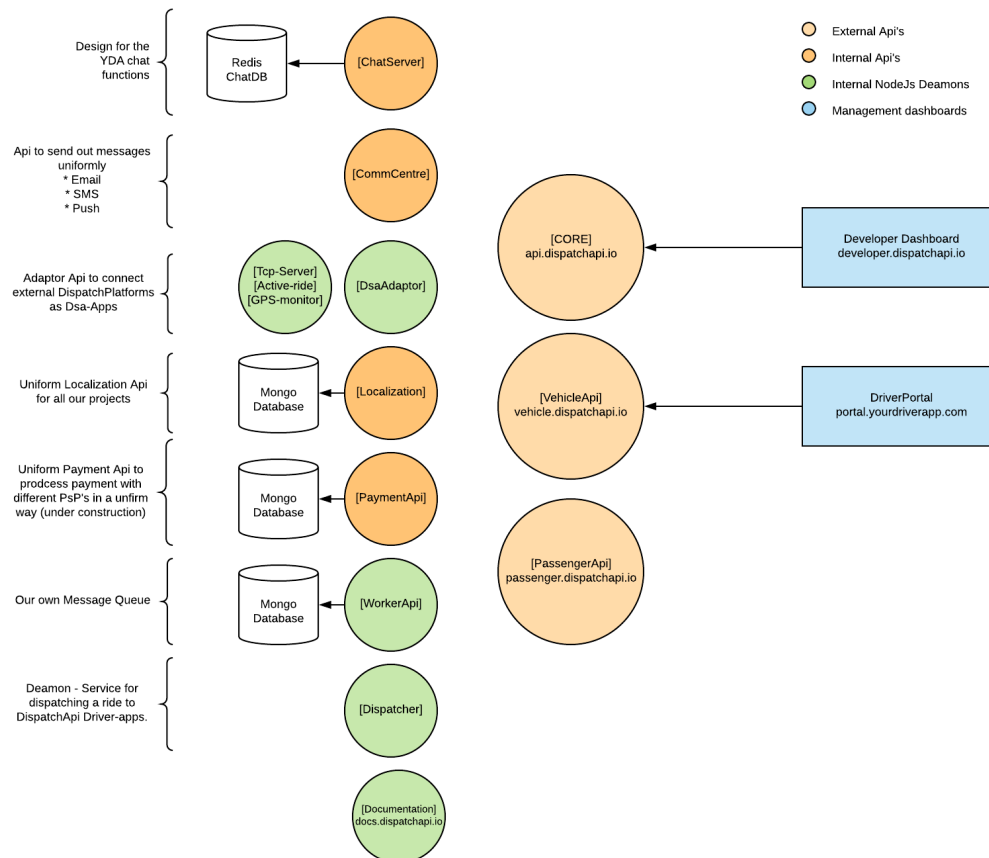


Fig. 3.1 Current System Architecture provided by taxiID

Integration of the backend would mean that the core system would contain the price calculation system as a component, separation of the backend would mean that the backend would be set up as a separate service. Integration of the frontend would mean that the DriverPortal would contain the views to manipulate information in the backend, separation of the frontend would mean that the views would be delivered from a separate service, or from within the backend service as a single project. Building the TPS frontend and backend into one single project would be in conflict with this three-tier pattern, which would not be desired.

### 3.2.1 Monoliths and Microservices

What logically follows from implementing functionalities as components in monoliths is either duplication or dependence between larger systems. This contradicts an important

principle of software engineering; don't repeat yourself (DRY), or limits scalability and independence of deployment. The legacy system has implemented its price calculation system in this manner, now facing difficulties of expanding the functionality to new projects. If the previous price calculation system was implemented as a microservice, it could have been reused or deployed as a second separate price calculation system for YDA. Fiar points of criticism have been made in regard to microservices. Jan Stenberg has pointed out that microservices are information barriers in [8], meaning that the process of implementing a new system is degraded by the sense of ownership of specific services by developers. Technical downsides that have been discussed in general are: latency, testing, deployment, and message formats. As with most technological decisions, whether a microservice is desired depends on the problem that it meant to solve. When should a system be integrated or separated? It is a careful balancing between efficiency of satisfying dependencies keeping a separation of concern, and duplication of functionalities. When a single responsibility is handled by one service, it may be classified as a microservice. Each microservice is responsible for managing and containing state to enable users who would like to use the system to be authenticated and authorized. Advantages of a microservice are the fact that a microservice is a self-contained and has a naturally modular structure. But authentication and authorization must be handled by the microservice itself, unless state is shared amongst services, which would eliminate the reason to use a microservice in the first place. In the current system architecture, different services implement different authentication methods, store different pieces of information of different users. Authorization is managed by sending extra headers for each crucial piece of information, this is clarified in Appendix A, chapter 3.4. For example: company information, application information, and user information is all sent in separate headers. When the amount of services that are added to the architecture increases, the mount of information that is no longer centralized increases with it.

### 3.3 Information Dependencies

Mapping the information dependencies is important while making architectural decisions. Being able to satisfy information dependencies dictates whether a microservice can be used for example. Trip prices are calculated on company level. A company may have several applications that can make use of company pricing rules.

- User data
- Vehicle data
- etc..

## 3.4 Authentication and Authorization

Mobile applications should be able to make requests, just like the portals that are to be developed. But portal users make use of the microservice in a different way. Mobile apps merely request prices of products, based on the rules that group admins define through the portal. To make sure that only the portal users have the right to mutate their data, users have to be authenticated and authorized within the microservice. Identity management becomes a problem if data duplication is not desired. If a user makes a direct request to the microservice, the credentials have to be compared to user data in a database. To prevent duplication, the microservice could be connected to the database that is used by the core system. But this makes the microservice less decoupled, and directly contradicts the desire to separate concerns. Four examples demonstrate this problem:

- Example 1: The microservice authenticates and authorizes users all by itself, managing sessions and storing user data in its database.
- Example 2: The microservice connects to an existing databases to acquire the required information about the user.
- Example 3: The core system authenticates the user and provides a token that can be verified by the microservice, containing user identity.
- Example 4: A separate service is used for authentication and authorization so that the core system is not involved at all.

In the first example, the microservice seems to work independently, because it has knowledge about the users identity without making requests to adjacent systems, or connecting to external databases. But this is not true. If data about the user is mutated in the core system, the microservice needs to be notified or synced. This greatly hinders scaling and makes it harder to keep data consistent. Example two solves the inconsistency part by connecting to the central database that holds user data, but contradicts the strive for encapsulation.

### 3.4.1 JSON Web Tokens

Example three entirely removes the database connection to any user data. This is possible when a JSON Web Token (JWT) is used. A JWT may be signed with a cryptographic algorithm or even a public/private key pair using RSA. After the user enters valid credentials, the core system validates the credentials by comparing them with user data in the database. The core system signs a token that with a secret that is known by the microservice. The token

consists of three parts, separated by a fullstop. The first part (header) of the token contains information about the hashing algorithm that was used to encrypt the payload. This part is Base64Url encoded. The payload itself contains information stored in JSON format:

kks32: Show diagram with hierarchy of companies and apps



```
{
  "companyId": "59ea0846f1fea03858e16311",
  "daAppInstallId": "599d39b67c4cae5f11475e93",
  "iat": 1521729818,
  "exp": 1521816218,
  "aud": "tps.dispatchapi.io",
  "iss": "api.dispatchapi.io",
  "sub": "getPrices"
}
```

The identity of the user is stored in the payload that can only be revealed by whoever holds the secret with which it was signed. Then the message can be verified using the third part of the token, which is the signature. This verification step prevents tampering with the payload.

### 3.4.2 oAuth 2.0

Example four delegates managing user identity to a separate authentication service that, similar to the pricing microservice, has its own single task of authenticating users. OAuth 2.0 is a protocol that has been designed to allow third-party apps to grant access to an HTTP service on behalf of the resource owner. This behaviour could be utilized to allow users to make use of services within the architecture, controlled by a single service, stored in a single token. A proposal was made in the Pregame document to combine oAuth with JWT and an API Gateway to introduce an automated authentication flow with a single token, instead of sending multiple headers, see Appendix A.





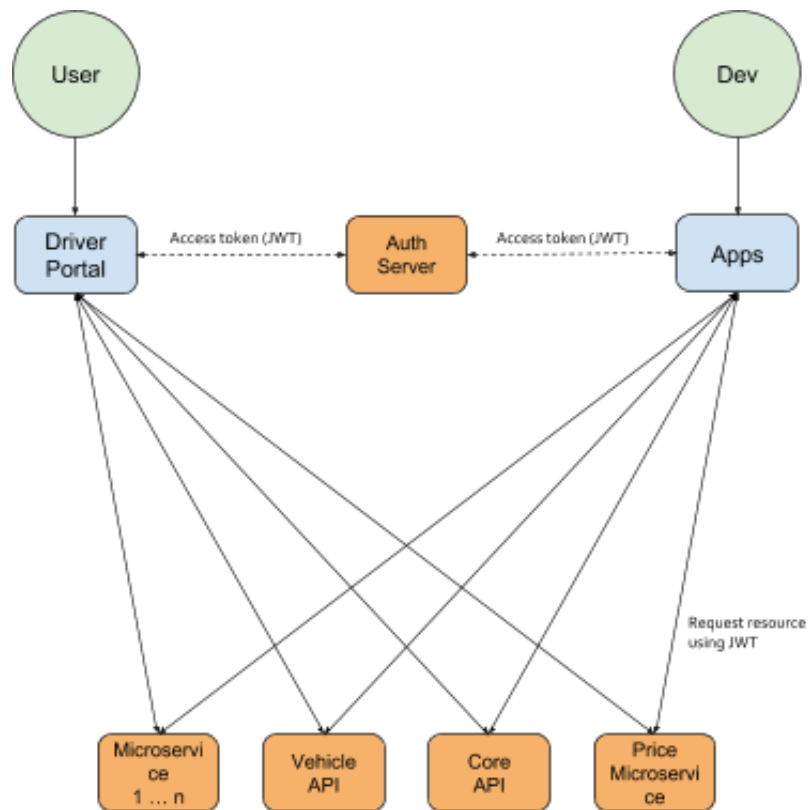


Fig. 3.3 OAuth with stateless JWT token requests

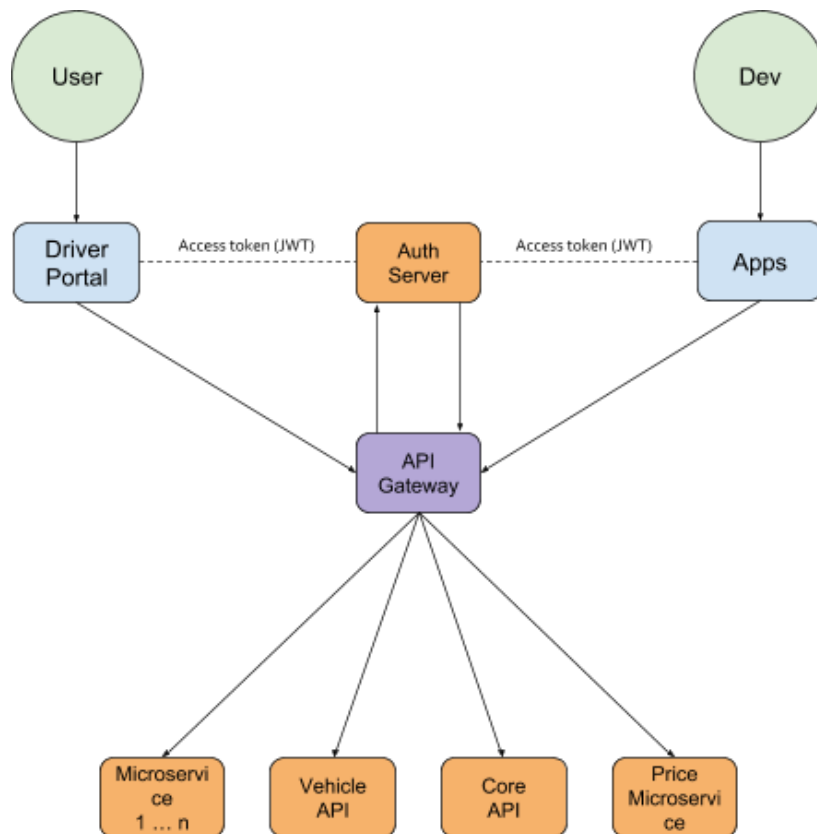


Fig. 3.4 API Gateway

### 3.4.3 API Gateway

Another common structure that allows services to be used by external agents is the API Gateway. It allows for a central middleware in which authentication and authorization is handled, where the microservices are shielded from public access, and all communication is established through the API Gateway [4].

Next to authentication, the gateway could optimize the endpoints so that no multiple requests are needed from external agents to gather different types of resources. These calls could be made internally to the microservices behind the gateway. This also opens the possibility to freely change the microservices without changing the public endpoints exposed by the gateway, and even offers slow or instant transitions to different versions of microservices.

The different proposals explain the improvements they may bring over some system. But the advice given is not tied to this project, instead to the entire Dispatch API. It's advised to have a constructive dialogue about the future of the company, and the way it's planning to scale. One could put a API Gateway in front of a monolithic app to help with transitioning to a microservice-oriented app.

## 3.5 Suitability of Methods and Technologies

kks32: Write chapter about methods and technologies

- Node JS, PHP, MongoDB, MySQL, Microservice, Loopback, GraphQL,
- Slides and proposals
- Mocha, Buddy-Works, Circle CI, Typescript, Chai, Functional Programming



# Chapter 4

## Trip Price Calculation System

### 4.1 Introduction

This chapter clarifies which information should be present in the breakdown, what the logical flow that results into a price breakdown looks like, which problems have to be solved

- What does a breakdown look like? - What information is required for a price calculation: vehicle types, passenger count, timeframes, locations - Which steps are taken before the calculation is finished: priority, - Which solutions are deemed realistic?

### 4.2 Breakdown

To make sure that the transition from the previous price calculation system to the new one is perfectly seamless, the responses that the apps expect should have the exact same format. An array containing breakdowns for each product that is associated with the triggered pricing rule, as can be seen in 4.1.

```
1 [
2   {
3     "vehicleType": "estate",
4     "maxPassengers": 4,
5     "fixedPrice": true,
6     "price": {
7       "breakdown": {
8         "route": 8300,
9         "toll": 0,
10        "parking": 0,
11        "waiting": 0,
12        "discount": -1650
13      },
14      "currency": "EUR",
15      "total": 6650,
16      "tax": {
```

```
17     "amount": 400,  
18     "percentage": 6  
19   }  
20 }  
21 },  
22 ...  
23 ]
```

Listing 4.1 Array of products with pricing

It is important to mention that VAT is included, and does not add up to the total price. All amounts are displayed in cents.

## 4.3 Timeframes

Next to the three dimensions of space, time will play a role in determining whether a rule has matched. The implementation of this concept should preferably offer enough freedom in the future, and should not be tailored toward one specific entity relation. Being able to reuse the timeframe entity improves maintainability of the system. The requirements state that the user must be able to define a start and end time, the days on which the times are active, and the start and end date of the timeframe. The timeframe is used to describe when rules or discounts are active. If, for example, a discount should be active during night of New Years Eve, between 23h and 5h, this description of a timeframe is already troublesome.

### 4.3.1 Conventional Approach

The current taxiID system solved this issue by storing multiple child entities with time information for each day, where the time inputs ranged from 00:00 up until has implemented this straight forward solution of storing the begin and end of some event as timestamps as child entities to some parent timeframe. These child windows can be iterated or queried to see whether one contains the specified timestamp.

- In what ways can timeframes be data modelled for a database? - TIME - Timestamp

### 4.3.2 Bitmap

## 4.4 Data Model

## 4.5 Logical Flow

- authentication - extracting companyId and daAppInstallId - extracting departure and destination coordinates - using directions service to acquire distance and duration - converting to minutes and km - executing query - find daAppInstall model with matching companyId and daAppInstallId - find company country - find matching discounts ordered by priority - enabled - timeframes - departure - destination - find matching rules ordered by priority - enabled - timeframes - departure - destination - get pricing information of all products for highest priority rule

A tier price system, that calculates fixed prices based cascading thresholds, and a dynamic pricing system that calculates prices per distance unit and minute is a very specific problem that must be split up into sizable categories. The term 'distance unit' is used on purpose, as distances are measured using different metrics in various countries. Pricing rules should be constrained by time frames, making rules available only for some hours a day, or only on christmas for example. Rules should be specifiable per product as different vehicle types have different prices, but are included in the same pricing rules. Discounts may be calculated with the trip price, and VAT should be displayed in the price breakdown. Some additional requirements to the system may be added in later phases, as Scrum is used to manage work iterations (this fact is covered later in this chapter). The system should be accessible to other systems, meaning that applications that currently rely on the old system should be able to migrate to the new system. As the old system shouldn't be used for new applications, as it was not designed for this use case. The system should have a single responsibility, and should be autonomous in that regard.





# Chapter 5

## Proposed Portal Solution

### 5.1 Introduction

This chapter covers the actual implementation plan of connecting the pricing system with the portal frontend. How the system should behave under different circumstances, how the user is able to interact with the system. The YTA-portal should integrate the frontend that allows taxi company users to modify their pricing rules. This chapter aims to answer question number three, which aims to

### 5.2 Required Views

### 5.3 Methods and Techniques

### 5.4 Proposal Pricing Rules View

### 5.5

kks32: This is not researched yet, as it's covered in later sprints

How can the task of defining rules be as insightful as possible to the user?

1. Which views should exist, does a logical hierarchy exist among views?
2. How should locations be defined and managed by the user?
3. How should timeframes be handled in the interface?



# **Chapter 6**

## **Realization**

### **6.1 Introduction**

During the second phase, issues from the backlog were implemented in an iterative SCRUM process. In this chapter, the final realization of the project is evaluated. Findings and observations by considering the assumptions and limitations are discussed. During development, two main applications were written. The price calculation system, and the portal that enables users to manage pricing rules in the price calculation system.

### **6.2 Methods and Techniques**

In the first sprint, a project was set up in NodeJS using Typescript. The existing projects were built using Javascript, but Typescript is a much safer language, preventing bugs because the compiler can catch errors early on, warning programmers instead before the application crashes. Types also document code, expressing the intention of the programmer.

### **6.3 Sprint 1 - Dynamic Price Calculations**

A basic dynamic price calculation system was implemented in the first sprint, aiming to deliver a first version of the system, including fake data generators, validation of models, a single service to determine the distance and duration of a ride, rules that contain pricing information, a calculation that produces the total price of a ride using a companies rules, a formatter that produces an expected response, and tests for all of the functionalities.

## 6.4 Sprint 2 - Authentication and Authorization

Company pricing rules can be used by applications so that each application uses a subset of the pricing rules. For this reason, TPS requires two identifiers to make a price calculation using rules for a particular company application: a `companyId` and a `daAppInstallId`. JSON Web Tokens that are signed by the core system contain the identifiers in the payload, so that TPS can use the identifiers after decrypting the token. Companies have one country assigned by default, which determines the currency and VAT percentage. In the breakdown, the VAT percentage is calculated from the actual price, as VAT is included. Discounts are part of the breakdown, being a percentage of the route price, or a fixed price. On top of that, it is possible that a company application uses rules that are related to a debtor, instead of its own subset of rules. Finally, the project is deployed to a staging environment so that the system could be used by the applications in the staging environment.

## 6.5 Sprint 3 - Setting up the Portal

At this point the system is fully operational, but company and `daAppInstall` information has to be inserted in the database manually. An endpoint is made that inserts a full company setup into the database so that prices can be calculated with five products by default. No wireframes were made beforehand, making it a task for the current sprint being executed while setting up the portal project. Angular in conjunction with Covalents UI platform is used to make the user interface, consisting of an overview and detail page for products and pricing rules. The pricing rules overview shows pricing information for each product that a company has. Whenever a product is added, the pricing information for that new project is automatically added to each rule. Conversely, whenever a new rule is added, all the existing products get their pricing information added to the new rule. On top of that, threshold rules can be added or deleted for distances and durations, making this particular view very complex. This final task was only operational in the backend.

## 6.6 Sprint 4 - Expanding the Portal

Feedback was given by the product owner after each sprint, resulting into new requirements and modifications to requirements. A functionality was required that enabled the user to sort pricing rules and special rates (discounts), by dragging the rows in a table to the correct positions. Another requirement would enable products to be returned in the breakdown as 'on-meter' results. This meant that, whenever a destination or departure location was

undefined, the system would return products without a price, so that the apps could assign a price later, but would still be aware of the available products. A view was added that displayed all apps of a company, and a detail page was added in which rules and discounts could be associated with those apps. This detail page was made in three iterations. The text of all pages were replaced by references to the localization api as seen in figure ??.

#### Thresholds

Pricing rule priorities had to be draggable by the user, enabling ordering in a visual way. - thresholds continued - processing feedback - pricing rules overview, rules must be draggable to prioritize - priority must be maintained distinctly - app installations must be displayed - pricing rules and special rates must be enabled for specific app installs - allow on meter calculations - internationalization - timeframes -

## 6.7 Sprint 5 - Expanding the Portal

Threshold prices

## 6.8 Result



## **Chapter 7**

## **Conclusion**





## **Chapter 8**

### **Recommendations**



# References

- [1] U. T. Inc. (2011) The uber story. [Online]. Available: <https://www.uber.com/en-NL/our-story/>
- [2] J. R. Herring, “Implementation standard for geographic information - simple feature access - part 1: Common architecture,” May 2011. [Online]. Available: [http://portal.opengeospatial.org/files/?artifact\\_id=25355](http://portal.opengeospatial.org/files/?artifact_id=25355)
- [3] (2004) Geographic information – simple feature access – part 1: Common architecture. [Online]. Available: <https://www.iso.org/standard/40114.html>
- [4] J. R. Herring, “Implementation standard for geographic information - simple feature access - part 2: Sql option,” August 2018. [Online]. Available: [http://portal.opengeospatial.org/files/?artifact\\_id=25354](http://portal.opengeospatial.org/files/?artifact_id=25354)
- [5] (2018) Postgis 2.4.5dev manual. [Online]. Available: [https://postgis.net/docs/manual-2.4/using\\_postgis\\_dbmanagement.html#PostGIS\\_GeographyVSGeometry](https://postgis.net/docs/manual-2.4/using_postgis_dbmanagement.html#PostGIS_GeographyVSGeometry)
- [6] (2018) Mysql 5.7 reference manual - geometry class. [Online]. Available: <https://dev.mysql.com/doc/refman/5.7/en/gis-class-geometry.html>
- [7] I. K. Center. (2018) Three-tier architectures. [Online]. Available: [https://www.ibm.com/support/knowledgecenter/en/SSAW57\\_8.5.5/com.ibm.websphere.nd.doc/ae/covr\\_3-tier.html](https://www.ibm.com/support/knowledgecenter/en/SSAW57_8.5.5/com.ibm.websphere.nd.doc/ae/covr_3-tier.html)
- [8] J. Stenberg. (2014) Experiences from failing with microservices. [Online]. Available: <https://www.infoq.com/news/2014/08/failing-microservices>



# List of figures

1.1	Fixed Prices . . . . .	2
2.1	LatLngSphere . . . . .	7
2.2	Square . . . . .	11
3.1	Architecture . . . . .	16
3.2	Architecture . . . . .	20
3.3	Architecture . . . . .	21
3.4	Architecture . . . . .	22



## List of tables





# **Appendix A**

## **Pregame**

### **Pregame**

# Phase I - Pregame

Prices API - 05 Feb 18 t/m 30 Aug 18

betreft Afstudeerstage bij TaxilD

jaar 2018



Author: Stefan Schenk  
500600679

Tutor: Willem Brouwer

TaxilD  
05-02-2018 te Medemblik

Hogeschool van Amsterdam  
Software Engineering

# Index

<b>1. Introduction</b>	<b>3</b>
<b>2. Requirements Specification</b>	<b>4</b>
2.1. Purpose	4
2.2. Scope	4
2.2.1. Deliverables:	4
2.2.2. Impact:	4
2.2.3. Assumptions:	4
2.3. Stakeholders	4
2.4. Use Case Diagram	5
2.5. Requirements	6
2.5.1. Non-functional Requirements	6
2.5.2. Functional Requirements	6
2.6. Constraints	7
2.7. Definitions, Acronyms, and Abbreviations	7
2.8. Use Cases	8
<b>3. Definition</b>	<b>12</b>
3.1. Non-functional Requirements	12
3.2. Functional Requirements	12
3.2.1. Defining an Area	12
3.2.2. Requirements for Rules	13
3.2.3. Other Requirements	14
3.3. Architecture	14
3.4. Authentication and Authorization	15
3.5. Database	16
3.6. API	16
3.7. User Interface	16
3.8. Database Schema	16
3.9. Continuous Integration, Continuous Deployment & Testing	16
<b>4. Solution</b>	<b>18</b>
4.1. Non-functional Requirements	18
4.2. Functional Requirements	18
4.2.1. Trip Price Calculation	19
4.2.2. Defining Price Rules	20
4.2.3. Defining Locations	20
4.2.4. Defining Timeframes	21
4.2.5. Defining Discounts	21
4.2.6. Defining Debtors	21
4.2.7. Defining Vehicle Types	21
4.3. Architecture	22

4.4. Authentication and Authorization	22
4.4.1. Proposal oauth 2.0 refactor	22
4.4.2. Jwt token format proposal	23
4.4.3. Proposal API Gateway	24
4.5. Database	25
4.5.1. OpenGIS Compatible databases	26
4.5.2. OpenGIS Incompatible databases	27
4.5.3. Performance and Clustering Trade-offs	28
4.6. API	28
4.6.1. Required Endpoints	28
4.6.2. Express VS Loopback	28
4.7. Database Schema	28
4.7.1. Relational Database	29
4.7.2. Non-Relational Database	29
4.8. User Interface	30
4.9. Continuous Integration, Continuous Deployment & Testing	30
4.10. Testing	31
<b>5. Conclusion</b>	<b>32</b>
5.1. Frontend	32
5.2. Backend	32
5.3. Functionalities	32
5.4. Authentication and Authorization	33
5.5. Database	33
5.6. User Interface	33
<b>6. References</b>	<b>34</b>

# 1. Introduction

The pregame phase concerns about planning and architecture, also called sprint zero, which is usually adopted when scrum is used as a business process for practical purposes. The first step is creating the backlog - a list with things that have to be implemented during the game phase. Because scrum is not fully adopted within the project team, this document contains another chapter that translates the requirements, written by the product owner and one developer of the team (in chapter 2), to a problem definition (in chapter 3), whereafter an architectural solution is presented (in chapter 4).

## 2. Requirements Specification

This section introduces the requirements set for the trip price calculation system written by one developer of the team and the product owner.

### 2.1. Purpose

YDA (YourDriverApp) requires a pricing calculation functionality that is similar to the existing taxiID implementation. All functionalities within the current system align with the clients wishes, but some features bring certain difficulties along, for example: region names are too vague for specific database queries. Some features could be abstracted so more possibilities can be implemented, some features are still unimplemented, and some features could be improved along the way.

### 2.2. Scope

#### 2.2.1. Deliverables:

1. A trip price calculation microservice or module in the dispatch api platform (for simplicity will be referred to as microservice).
2. The communication between other services within the architecture, and alignment of changes to support this new microservice.
3. Documentation describing the API.
4. A user interface in the driver portal wherein the User can define trip prices that exist in the current system.
5. A English user manual explaining the user interface.

#### 2.2.2. Impact:

1. No costs other than a possible substitution for Google services tackling the problem of inaccurate GPS to road mappings.
2. Small strain on developers for supporting integration and possible modifications within the system architecture.

#### 2.2.3. Assumptions:

1. NodeJS will be used to develop systems, unless a very good reason is given to deviate from this established technology.
2. MongoDB is used in many projects, and therefore is preferable over other RDB systems.
3. Authorization will be handled, and is being discussed internally.
4. GPS coordinates will be provided in addition to ambiguous place descriptors on every price calculation.

### 2.3. Stakeholders

Name	Role	Expectations
YourDriverApp Group Admin	End user	A price calculation system.
taxiID Account Admin	End user	Seamless transition without loss of functionalities from TaxiID price calculations to the new system

Driver App User	End user	No changes
Passenger	End user	No changes
Project team	Project members	Well documented easy to maintain and easy to extend system
Product Owner	Project manager	A working version at the end of every sprint with added functionalities each iteration

## 2.4. Use Case Diagram

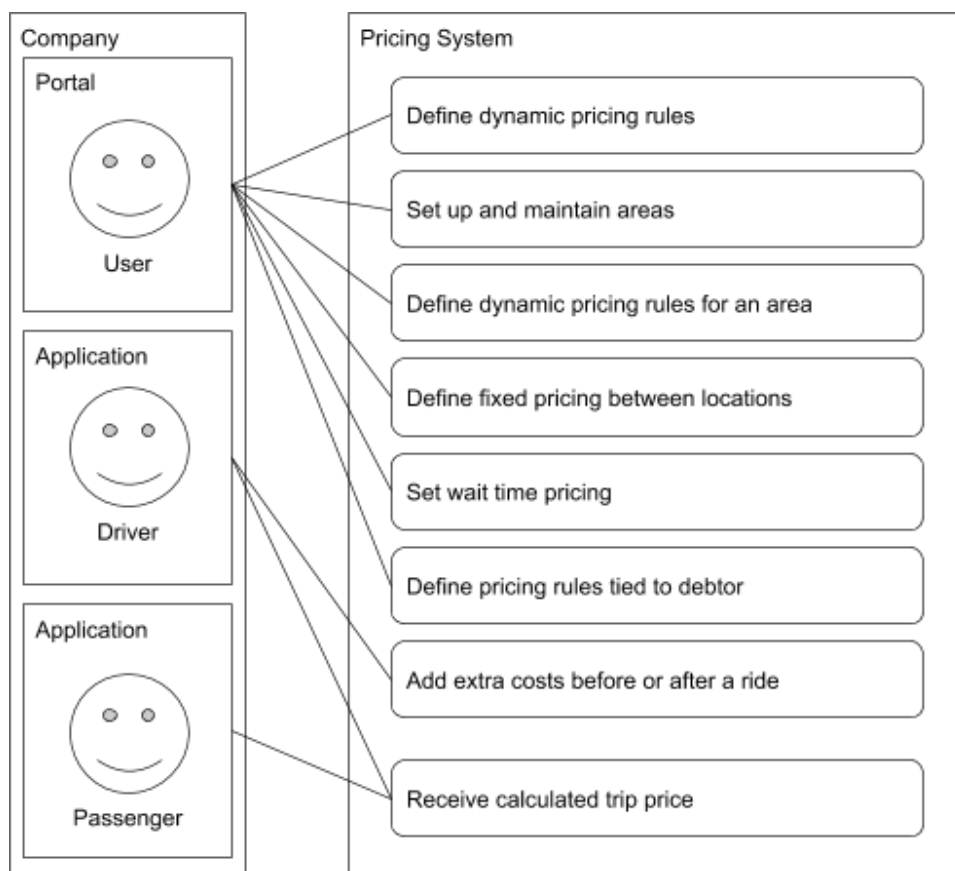


Image 2.4.1 - Use case diagram.

## 2.5. Requirements

### 2.5.1. Non-functional Requirements

ID	Non-functional Requirement
<b>NFR1</b>	For a logged in driver portal user (yourdriverapp.com or white labeled build) the solution should be seamlessly integrated in the portal.
<b>NFR2</b>	A logged in taxiID partner portal user should be able to set my rates without having to log in again. Visual integration is not important but the brand yourdriverapp.com should not be visible.
<b>NFR3</b>	The prices should be attached to a DaAppInstall.

### 2.5.2. Functional Requirements

ID	Functional Requirement
<b>FR1</b>	<p>A user should be able to set up and maintain dynamic rules for a calculation based travel time and travel distance between a pickup and drop off position.</p> <p>This price should be calculated taking into account:</p> <ol style="list-style-type: none"><li>1. Starting rate</li><li>2. Rate per km / mile - it should be possible to add at least 5 user defined segments (i.e. a price for the first km, a lower rate for km 2 to 3, an even lower rate for every km after 4 km</li><li>3. Rate per minute - it should be possible to add at least 5 user defined segments (i.e. a price for the first travel minute, a lower rate for minute 2 to 3, an even lower rate for every minute after 4</li><li>4. This calculation may be done in advance based on online route planner service calculations or afterwards based on trip data from the driver app.</li></ol>
<b>FR2</b>	A user can define a price per minute for waiting time, the spent wait time can be sent by the driver.
<b>FR3</b>	As a user I want to select areas from a predefined list to set up fixed price calculations.
<b>FR4</b>	A user should be able to set up and maintain areas for a company. Examples of areas are: neighborhood, province, region, city, hospital, airport, train stations, hotels. We should have some types/tags predefined.
<b>FR5</b>	A user should be able to set up and maintain distinct calculations based on travel time and travel distance for different areas defined by the user.
<b>FR6</b>	A user can define fixed prices based on specific clients, potentially tied to a debtor. This is going to be based on polygons/areas too.
<b>FR7</b>	<p>A driver can add positive or negative additions to the cost of the ride at any point in time.</p> <ul style="list-style-type: none"><li>- Percentage (discount)</li></ul>



	<ul style="list-style-type: none"> <li>- Driver defined (toll, parking, other)</li> <li>- Variable (waiting time - it has to be calculated inside the system, from an input of time)</li> </ul>
<b>FR8</b>	It should be possible to set up a price with time constraints only (hire a limo) - this is just a dynamic rule
<b>FR9</b>	A user can have pricing rules based on different services than Google Maps. Defined per rule.

## 2.6. Constraints

As stated in the scope, the system that is to be implemented will either be implemented as a microservice or a module. In the latter case, the existing and adjacent systems will make way for the new module. This adds extra requirements for the new system to be integratable.

## 2.7. Definitions, Acronyms, and Abbreviations

<b>Bulk:</b>	Either in the context of time or distance, a threshold that can be set after which the price per unit will be cheaper (or more expensive).
<b>CD:</b>	Continuous Delivery / Deployment.
<b>CI:</b>	Continuous Integration.
<b>Company:</b>	A company that owns Applications.
<b>DaAppInstall</b>	An application installation.
<b>Debtor:</b>	A person or company responsible for the payment of a ride, on upon which the pricing can depend.
<b>Driver Portal:</b>	Portal that brings information from diverse sources.
<b>Discounts:</b>	A discount that is either a percentage, fixed amount or reference to rule containing prices.
<b>Location:</b>	A zip code or geometric location.
<b>ORM:</b>	Object Relational Mapping.
<b>User:</b>	A person, group or company that owns applications.
<b>Passenger:</b>	Uses an Application to order a taxi ride.
<b>Product / Application:</b>	An application bought by the User to which data is tied.
<b>Pricing Rule:</b>	A body of information that can be triggered when a ride is selected that matches the destination, departure and perhaps other variables, which contains pricing information about that ride depending on distance, time and other parameters.
<b>taxiID Partner Portal</b>	Portal that brings information from taxiID sources.
<b>Timeframe:</b>	A collection of start and end times + days of the week.
<b>Zones / Regions:</b>	Polygons drawn on a map.
<b>Core API:</b>	Available through Developer Dashboard (developer.dispatch.io).
<b>Passenger API:</b>	Available through Passenger App.
<b>Vehicle API:</b>	Available through DriverPortal (portal.yourdriverapp.com).

## 2.8. Use Cases

The following use cases are describing a passenger who orders a ride, for which a price is calculated by the API. The primary actor, preconditions and other information is omitted for conciseness.

The first step for every case is the following:

1. The passenger books a ride where properties are sent to the API unless mentioned otherwise:
  - a. Departure location
  - b. Destination location
  - c. Pickup datetime
  - d. Vehicle Type
  - e. DaAppInstall token
  - f. Debtor identifier
  - g. Number of passengers

ID	Use Case
1	Passenger app sends debtor identifier, a pricing rule is found, discount is found
2	Passenger app doesn't send debtor identifier, a pricing rule is found, discount is found
3	Passenger app doesn't send debtor identifier, a pricing rule is found, multiple discount are found
4	Passenger app doesn't send debtor identifier, a pricing rule is found, no discount are found
5	Passenger app doesn't send debtor identifier, a pricing rule isn't found, discount is found
6	Departure contains a point that matches with an area in a rule, there are multiple rules tied to the location
7	Departure location is contained locations A1 and B1, Destination location is contained in locations A2 and B2, therefore two rules are matched

ID	1
Description	Passenger app sends debtor identifier, a pricing rule is found, discount is found
Basic Flow	<ol style="list-style-type: none"><li>1. Debtor identifier is sent to the API</li><li>2. The API checks if a debtor identifier is sent, and it exists in the database</li><li>3. The API tries to match the pricing rules that are tied to the debtor by:<ol style="list-style-type: none"><li>a. Departure location</li><li>b. Destination location</li><li>c. Ride time</li></ol></li><li>4. A rule is found, the API tries to find a discount that is tied to the debtor based on:<ol style="list-style-type: none"><li>a. Departure location</li><li>b. Destination location</li><li>c. Ride time</li></ol></li></ol>

	<ol style="list-style-type: none"> <li>5. A discount rule is found</li> <li>6. The fixed price is calculated with the discount</li> </ol>
--	---

ID	2
Description	Passenger app doesn't send debtor identifier, a pricing rule is found, discount is found
Basic Flow	<ol style="list-style-type: none"> <li>1. Debtor identifier is not sent to the API</li> <li>2. The API checks if a debtor identifier is sent, it isn't</li> <li>3. The API tries to match general pricing rules tied to the company by: <ol style="list-style-type: none"> <li>a. Departure location</li> <li>b. Destination location</li> <li>c. Current time</li> </ol> </li> <li>4. A pricing rule is found, the API checks whether a discount is available that matches: <ol style="list-style-type: none"> <li>a. Departure location</li> <li>b. Destination location</li> <li>c. Ride time</li> </ol> </li> <li>5. A discount is found</li> <li>6. The fixed price is calculated with the discount</li> </ol>

ID	3
Description	Passenger app doesn't send debtor identifier, a pricing rule is found, multiple discount are found
Basic Flow	<ol style="list-style-type: none"> <li>1. The API tries to match general pricing rules tied to the company by: <ol style="list-style-type: none"> <li>a. Departure location</li> <li>b. Destination location</li> <li>c. Current time</li> </ol> </li> <li>2. A pricing rule is found, the API checks whether a discount is available that matches: <ol style="list-style-type: none"> <li>a. Departure location</li> <li>b. Destination location</li> <li>c. Ride time</li> </ol> </li> <li>3. Multiple discount are found</li> <li>4. The discount rule with the highest precedence is taken</li> <li>5. The fixed price is calculated with the discount</li> </ol>

ID	4
Description	Passenger app doesn't send debtor identifier, a pricing rule is found, no discount are found
Basic Flow	<ol style="list-style-type: none"> <li>1. The API tries to match general pricing rules tied to the company by: <ol style="list-style-type: none"> <li>a. Departure location</li> <li>b. Destination location</li> <li>c. Current time</li> </ol> </li> <li>2. A pricing rule is found, the API checks whether a discount is available that matches:</li> </ol>

	<ol style="list-style-type: none"> <li>a. Departure location</li> <li>b. Destination location</li> <li>c. Ride time</li> </ol> <ol style="list-style-type: none"> <li>3. No discount is found</li> <li>4. The fixed price is calculated</li> </ol>
--	--

ID	5
Description	Passenger app doesn't send debtor identifier, a pricing rule isn't found, discount is found
Basic Flow	<ol style="list-style-type: none"> <li>1. The API tries to match general pricing rules tied to the company by: <ol style="list-style-type: none"> <li>a. Departure location</li> <li>b. Destination location</li> <li>c. Current time</li> </ol> </li> <li>2. A pricing rule isn't found, the API tries to find a dynamic price rule</li> <li>3. A dynamic price rule is found, the API checks whether a discount is available that matches: <ol style="list-style-type: none"> <li>a. Departure location</li> <li>b. Destination location</li> <li>c. Ride time</li> </ol> </li> <li>4. A discount rule is found</li> <li>5. The fixed price is calculated with the discount</li> </ol>

ID	6
Description	Departure contains a point that matches with an area in a rule, there are multiple rules tied to the location
Basic Flow	<ol style="list-style-type: none"> <li>1. The API tries to match general pricing rules tied to the company by: <ol style="list-style-type: none"> <li>a. Departure location <ol style="list-style-type: none"> <li>i. The point is found in the area in the database</li> </ol> </li> <li>b. Destination location <ol style="list-style-type: none"> <li>i. The point is found in the area in the database</li> </ol> </li> <li>c. Ride time <ol style="list-style-type: none"> <li>i. The timeframe contains this ride time</li> </ol> </li> </ol> </li> <li>2. Multiple rules are found that match locations and timeframe <ol style="list-style-type: none"> <li>a. The rule with the highest precedence (highest number) is picked to calculate the price</li> </ol> </li> <li>3. A discount is not found, the price is calculated</li> </ol>

ID	7
Description	Departure location is contained locations A1 and B1, Destination location is contained in locations A2 and B2, therefore two rules are matched
Basic Flow	<ol style="list-style-type: none"> <li>1. The API tries to match general pricing rules tied to the company:</li> </ol>

	<ul style="list-style-type: none"> <li>a. Departure location <ul style="list-style-type: none"> <li>i. The gps location is found in polygon collections of rule A and B</li> </ul> </li> <li>b. Destination location <ul style="list-style-type: none"> <li>i. The gps destination location is found in rule A and B</li> </ul> </li> <li>c. Ride time <ul style="list-style-type: none"> <li>i. The timeframe contains this ride time</li> </ul> </li> </ul> <ul style="list-style-type: none"> <li>2. Multiple rules are found <ul style="list-style-type: none"> <li>a. The rule with the highest precedence is picked to calculate the price (optionally the precedence can be set on the location level, from which an average can be used to determine the rule precedence)</li> </ul> </li> <li>3. A discount is not found, the price is calculated</li> </ul>
--	---

## 3. Definition

The requirements are written in a vague way as the user would describe his or her wishes. The most important question that must be answered before the development phase is commenced is: are the requirements achievable tasks, and can they be translated to backlog tasks available to be assembled to a sprint backlog? This will be researched in chapter four, before research can be conducted, the problem must be well defined, which is the purpose of this chapter.

### 3.1. Non-functional Requirements

A user who is logged in on yourdriverapp.com or a white labeled build, the solution should be readily available. The most straightforward answer would be to directly integrate the frontend into yourdriverapp. The requirements state that a taxiID partner should also be able to use the frontend. This means that multiple frontends should be developed for multiple external portals that plan to make use of the system, or that some solution should be developed that integrates in different external portals seamlessly, for example: using iframes or objects, as visual integration is not important as long as the brand (yourdriverapp) is not visible. The requirements also state that a logged in taxiID user should not be required to log in again, this directly demands that a user is authenticated and authorized from any external frontend to the prices system.


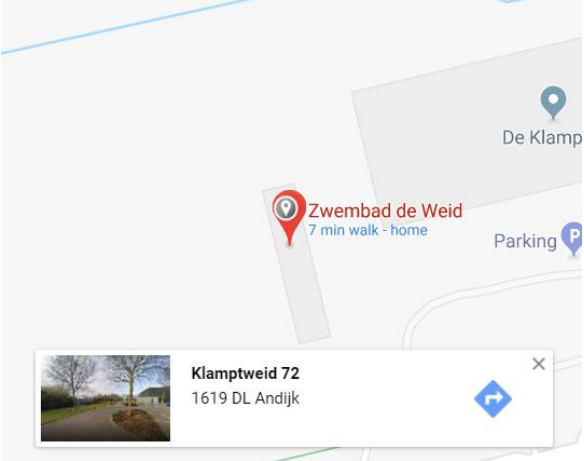
### 3.2. Functional Requirements

#### 3.2.1. Defining an Area

As most FR's depend on the term "area", it is a top priority to define what an area is. It's important to define locations in an unambiguous way so that no mistakes can be made like: selecting an area that is called the same. In some third-world countries, zip codes are not available, and area names can be ambiguously defined. Take for example "Third Main Street", a street name that may be used in thousands of distinct locations around the world. Therefore a different representation must be implemented for specific and general locations.

An area is a collection of 3 or more coordinate pairs on a geographical map. This definition of an area is precise, unambiguous and easy to use in compare in computer programs. A single point may match another single point if it's the exact same point. A point may be sitting on top of a line or is contained within an area. The only other option is the negation of these statements. Because use cases for lines will be non-existent, points and areas are the proper candidates for spatial queries.

The requirements state that a user must be able to define locations, or that he should be able to select predefined locations. It would be extremely easy for a user to search for a city, be able to import the polygon from some external source, edit it, save it, and perhaps even share it with other companies. A user should be able to find his own defined locations easily, or even distinguish between different types by tagging them.

Countries with advanced zip code systems	Countries without zip codes
 <p>Collection of zip codes to define an area</p>	 <p>Polygon on a map to define an area</p>
 <p>A single point defined by street,street nr, zip code city name</p>	 <p>A set of gps coordinates with a range to define a single point</p>

### 3.2.2. Requirements for Rules

The requirements state that users should be able to define dynamic prices, and that these dynamic prices should be tied to an area, or not. Dynamic prices can have zero values so that only a price per minute can be set. The requirements state that users should be able to define fixed prices from area to area. This implies that all types of pricing rules should be able to be tied to an area. The user should be able to assign different rules and discounts to a debtor, the same holds for DaApplnInstalls. It should be possible to define the timeframe in which rules hold as well.

### 3.2.3. Other Requirements

The user should be able to specify a price per minute that a driver has to wait for the passenger. The driver should also be able to add additions, additional costs, discounts or the amount of minutes that he waited for the passenger. Some additions must be expressed in percentages, continuous or discrete values. The user should also be able to specify the service that calculates the route of a trip.

## 3.3. Architecture

The existing architecture is shown below in image 3.3.1. The colored circle represents the change while the less colored shapes visualize the current state of the architecture.

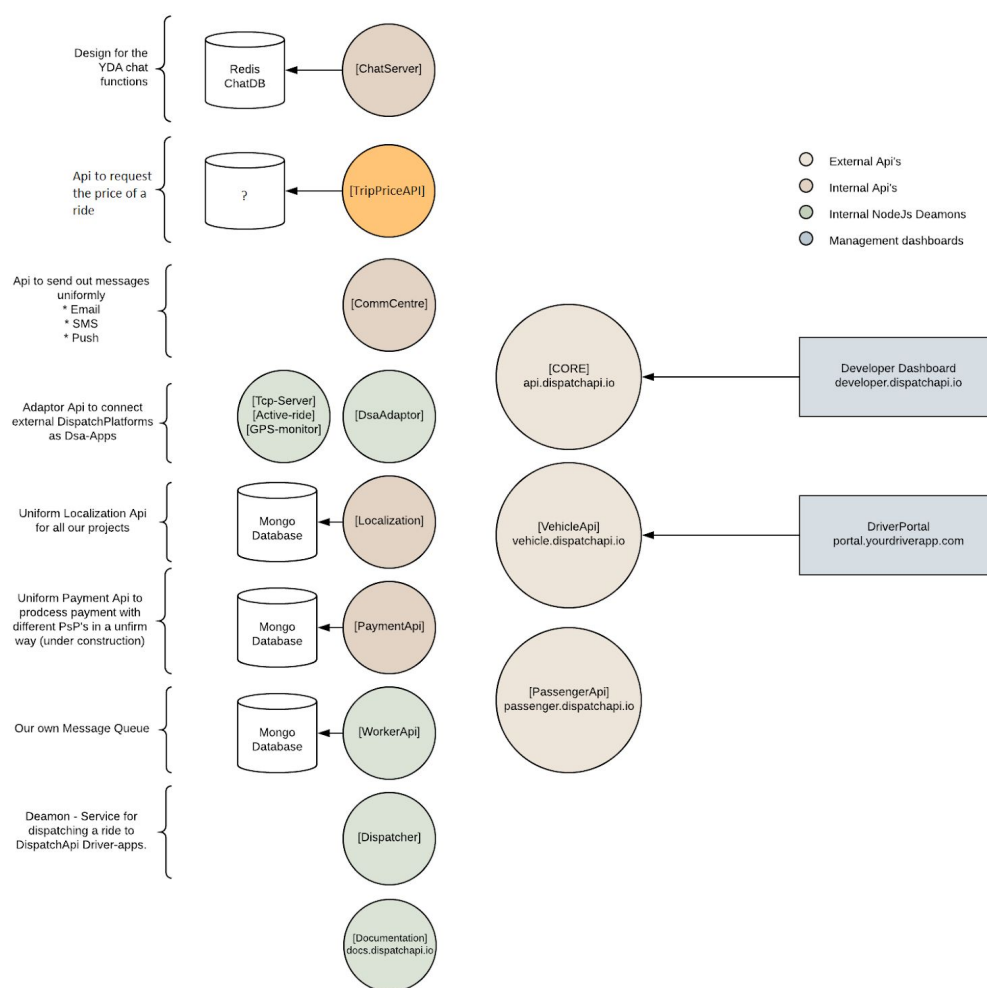


Image 3.3.1 - Current system architecture of the dispatch api.

The discussion that has risen from this image is whether the new system should be implemented as a microservice, or as a module in the existing project, see image 3.3.2. The orange and blue shapes can be in either state independently, meaning that four potential options exist, but are omitted for conciseness.



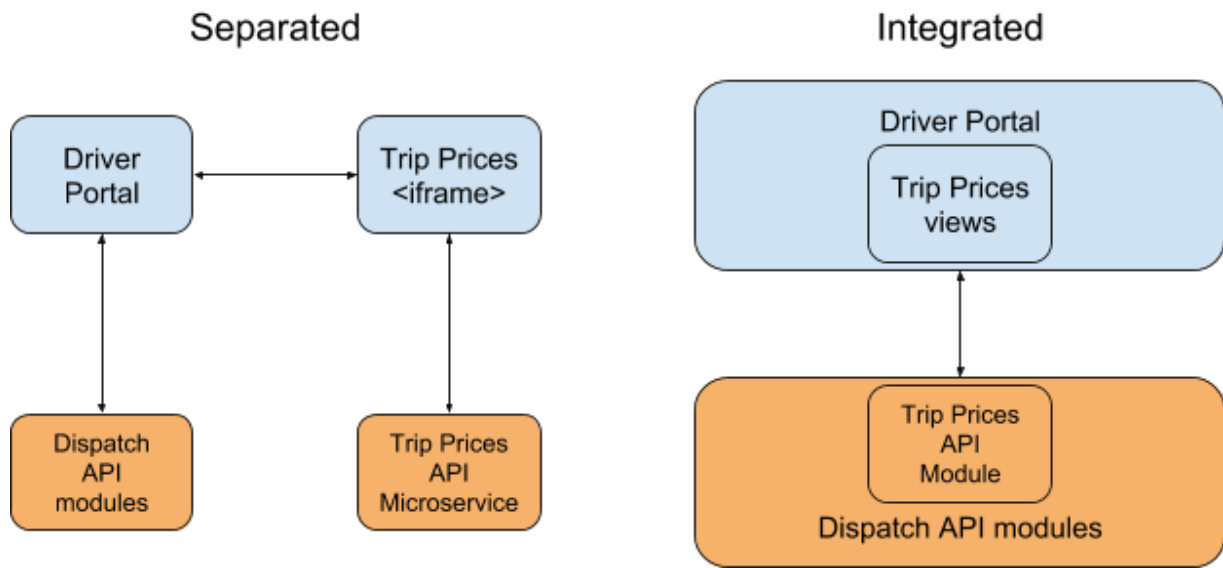


Image 3.3.2 - Separated and integrated frontend and or backend.

NFR2 either demands that a separate frontend is built, but this is not necessary if one frontend is built that can be integrated in existing pages, using the blue separated part of image 3.3.2.

### 3.4. Authentication and Authorization

The system must be autonomous and usable by agents from within and from outside the architecture it sits in. Therefore, authentication and authorization should be a matter of concern. It either changes the surrounding authentication solution, or implements a different solution to establish autonomy. For now, Drivers will make use of this service.

Marco Strijker has [documented](#) the three user types: Drivers, Passengers and Admins. Admins are a superset of Administrators, Developers and Organizations. All users log in with a username (email), password combination. After successfully logging in, an access token is provided which the user sends in the Authorization header to the corresponding API's.

Drivers log into the Vehicle API through the DriverPortal (or log in using their phone number in the Driver app), using headers:

1. **Authorization:** containing the access token
2. **X-Installation-Hash:** containing the authenticated installation of a Driver app.

Passengers log into the Passenger API through their Passenger app using headers:

1. **X-Access-Token:** containing the access token
2. **X-Company-Id:** containing encrypted company id with which

Admins are the developers working for TaxiID, developers are external developers, Organizations are external organizations, using the core API. This user type can install API apps by logging into the Developer Dashboard and granting permissions in a custom separate OAuth flow using headers:

1. **Authorization:** containing the access token

This project must have knowledge about who the user (Driver) is. Settings, prices, discounts and other required information to calculate a price are tied to the user.

### 3.5. Database

The only data that the system depends on is Master Data stored for each product, that the User will provide through the user interface. This system requires polygons to be drawn on a map that can be used to bivalently check whether a coordinate resides within it. For this reason it's important that the database supports complex spatial data, and performs well on complex queries. OpenGIS provides a way to define geometry models within MYSQL that is worth researching, [1] [2]. An ORM should be used to enable easy transitions between database systems.

### 3.6. API

Depending on the architectural choice described in chapter 2.1, the API will be integrated in an existing system, or will be set up from scratch. In the former case, extra models and endpoints must be added. In the latter case, a choice of framework and optional technologies must be made.

As Loopback is the framework that has been used extensively at TaxiID, this project could be an opportunity to test Loopback 4 in conjunction with Typescript for typesafe code. Alternatively Express or any other framework in conjunction with GraphQL could be interesting to look at.

### 3.7. User Interface

Just like the API, the App could be integrated or separated. The integrated solution considers the expansion of the existing Driver Portal, having the advantage of sharing resources efficiently and ensuring the exact same style. Alternatively the application could be developed independently, which could then be loaded into existing web pages using iframes or objects. Again, just like the API, the App is created from scratch if a separated solution is preferred, opening up the possibility to make use of the most modern techniques.

### 3.8. Database Schema

The schema's that will support the system should be concise and efficient naturally. Perhaps multiple databases should be used to support different data types, and therefore the schema's will look totally different. Therefore, this matter is succeeding the database topic.

### 3.9. Continuous Integration, Continuous Deployment & Testing

Lastly, Continuous Integration and Continuous Deployment may be utilized in early stages of development for the same reasons as Typescript and other new technologies could be trialled. TaxiID is a customer of Buddy.works, and therefore it may not be necessary to use other providers like: Jenkins, Travis CI, CircleCI or others.

Next to automated tests and linting, deployment may be automated upon successful integration. Heroku provides a free product that integrates easily with nearly all CI providers. Until the project is in production, Heroku can be used to make the product previewable for other developers or stakeholders.

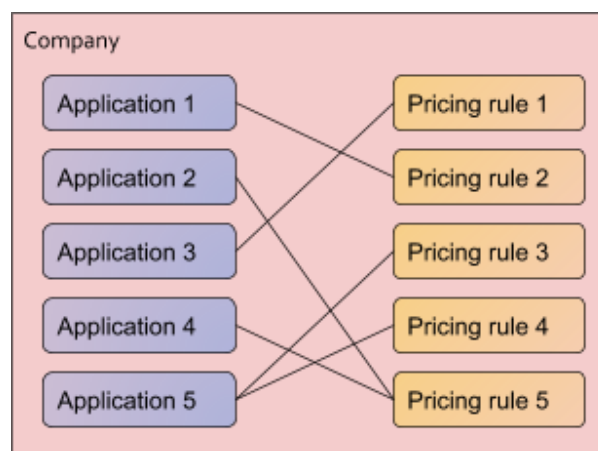
## 4. Solution

Now that the problems are well defined, research can be conducted to come up with a workable solution.

### 4.1. Non-functional Requirements

As stated in the NFR's, the frontend must be integrated in more than one application. This can be achieved using iframes or objects. More information on frontend and backend architecture is given in chapter 4.3.

The company's pricing rules should be attached to a DaAppInstall. This means that all applications within a company have their own subset of the pricing rules within that company:



*Image 4.1.1 - Company with applications and pricing rules.*

### 4.2. Functional Requirements

When we assume that the user is logged in, and has a company owning applications, several flows can be recognized: the trip price calculation, defining pricing rules, defining locations, defining discounts, defining timeframes. An important point to notice is how debtor should play a role in this calculation.

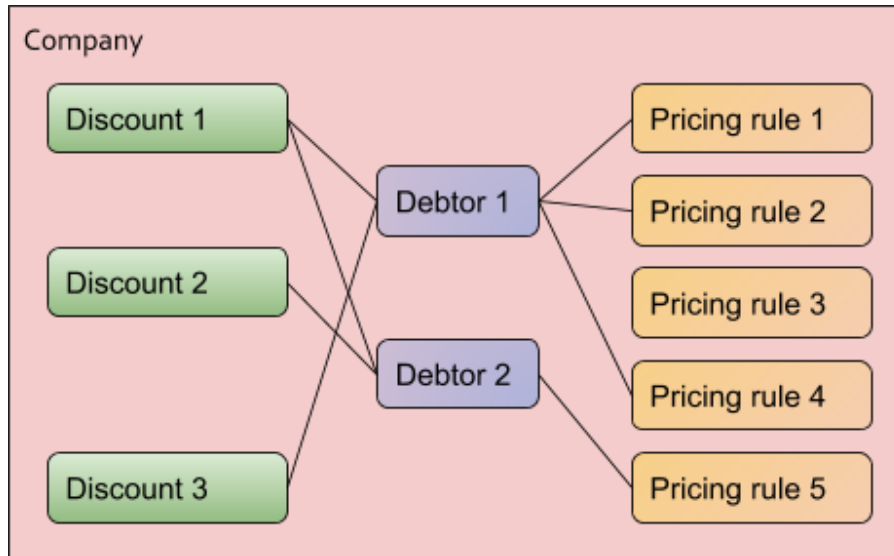


Image 4.2.1 - Debtors and the relation with pricing rules & discounts.

#### 4.2.1. Trip Price Calculation

1. APP: Passenger books a ride providing pickup location, drop off location, ride datetime, vehicle type array, amount of passengers, DaAppInstall token, (optional) debtor identifier. We will denote the fact that these properties fall within the criteria of pricing rules or discounts by using the word 'match'.
2. API: See if the company has a debtor, get debtor pricing rules and discounts, fallback to DaAppInstall rules.
3. API: If no debtor was linked, find DaAppInstall pricing rules and discounts.
4. API: Find pricing rules and discounts where the ride time is within the pricing rule timeframe.
5. API: Find pricing rules and discounts where the departure location contains the location provided by the user, and the rule location is of type:
  - a. Point
  - b. Polygon
6. API: Find pricing rules and discounts where the destination location contains the location provided by the user, and the rule location is of type:
  - a. Point
  - b. Polygon
7. API: If no rules were found, an error is returned.
8. API: To match points on points, we're gonna decrease the precision of the gps on queries.
9. API: Calculate prices depending on vehicle type and amount of passengers.
10. API: If more than one pricing rule was found, take the rule with highest precedence (highest number wins).
11. API: If more than one discount was found, take the rule with highest precedence (highest number wins).
12. API: Calculate discount.
13. API: Add additions defined by driver.app
14. API: Returns the trip price.

#### 4.2.2. Defining Price Rules

1. Portal: User accesses the pricing rule tab.
2. Portal: User adds or modifies a pricing rule.
3. Portal: User selects pricing rule type: (a or b).
  - a. Fixed: properties are provided
    - i. Pick up location is provided
    - ii. Drop off location is provided
    - iii. A price is provided
  - b. Dynamic: properties are provided
    - i. Start rate
    - ii. Minimum rate
    - iii. Waiting rate per minute
    - iv. Riding rate per minute
    - v. Riding rate for bulk minutes
    - vi. Riding rate per kilometer / mile
    - vii. Riding rate for bulk kilometers / miles
    - viii. Toggle: calculate each bulk using the bulk price, or only calculate the bulk units that have passed the threshold.
    - ix. Optional: A single location is provided for which these rules hold
4. Portal: User selects a timeframe for which the rule holds.
  - a. Timeframe can be disabled to enable the rule always
  - b. The timeframe editor view can be opened to make or modify a timeframe on the fly
5. Portal: User enables rule (activates it)
6. Portal: User can define a pricing rules for multiple debtors.
7. Portal: User can delete rules that have been created, except one fallback dynamic rule.

#### 4.2.3. Defining Locations

1. Portal: User accesses the locations tab.
2. Portal: User adds or modifies location.
3. Portal: There are two types of locations.
  - a. A single collection of points
  - b. A multipolygon / collection of polygons
4. Portal: Location can be defined and modified in two ways (a or b)
  - a. Single points can be added to a collection
    - i. By searching point of interests on Google Places API points will be suggested with fixed GPS coordinates
    - ii. Multiple points can be added to a point collection
  - b. An area can be added by drawing on a Google integrated Maps JS API
    - i. Areas can be added to the map by selecting from a predefined list
    - ii. Areas can be removed from the map
    - iii. Areas can be modified by dragging the edges of a polygon
    - iv. All areas can be stored as a single location (multipolygon)
5. Portal: User can delete custom locations that have been created.

#### 4.2.4. Defining Timeframes

1. Portal: User accesses timeframe tab.
2. Portal: User adds or modifies timeframe.
3. Portal: Timeframe can be defined in one way.
  - a. Optional: start date (absolute boundary)
  - b. Optional: end date (absolute boundary)
  - c. Hours enabled: (every single week)
    - i. Monday
    - ii. Tuesday
    - iii. Wednesday
    - iv. Thursday
    - v. Friday
    - vi. Saturday
    - vii. Sunday
4. Portal: User can delete timeframes, but only if they are not used by pricing rules, discounts or other entities.

#### 4.2.5. Defining Discounts

1. Portal: User accesses discounts tab.
2. Portal: User adds or modifies discounts.
3. Portal: User can link discount to multiple debtors.
4. Portal: User specifies properties:
  - a. Type: fixed or percentage
  - b. Amount
  - c. Optional: Timeframe
  - d. Optional: start location
  - e. Optional: end location
  - f. Toggle: Retour trip (present taxiID)

#### 4.2.6. Defining Debtors

1. Portal: User accesses debtors tab.
2. Portal: User can add or modify a debtor.
3. Portal: User can delete debtors.

#### 4.2.7. Defining Vehicle Types

1. Portal: User accesses vehicle types tab.
2. Portal: User can add or modify vehicle types.
  - a. User can copy a default vehicle type and modify properties of the copy, called a product:
    - i. Amount of passengers
    - ii. Image
    - iii. Name
  - b. User can store the product
3. Portal: User can delete products after a strict safety check (because they are potentially used in rules).

### 4.3. Architecture

The possibilities visualized in image 3.1.2. have great implications on adjacent systems, development time and maintainability. Table 4.1.1 shows the advantages (green) and downsides (red) of separation.

Frontend	Backend
Improves progressiveness of the entire architecture by incremental modernization steps.	
Improves maintainability by separation of concern.	
Brings the advantage of including the application in any portal in the future.	Improves testability by having small subsystems that can be isolated and tested while other systems can be relied upon.
May introduce a technical difficulty of presenting the view correctly into the portal.	May require extra http calls between services.
May hurt the visual style.	
Separation introduces a slight overhead because two separate views must be downloaded.	

*Table 4.1.1 - Pros and Cons of separation.*

After discussing the proposal to segregate this project from the existing Dispatch API, it is advised to implement the backend as a microservice, not as a module within the existing system because the only downside that was listed is trivial if the services are running on the same server. From the viewpoint of this project, it is also advised to separate the frontend using iframe, embed, object tags or some other solution.

### 4.4. Authentication and Authorization

A microservice architecture is an architectural style that focuses on loosely-coupled services, enabling continuous deployment of complex applications. Each microservice is responsible for managing and containing state that is used or exposed to other services that make use of the microservices, and must be authenticated and authorized to be able to use or request resources. In the present architecture, different services implement different authentication methods, store different information about different users. Authorization is managed by sending extra headers as described in chapter 2.2. By adding more services, the amount of authentication, authorization and user types will increase. For this reason it's profitable and even requested to investigate whether a better structure could be implemented.

#### 4.4.1. Proposal oauth 2.0 refactor

There exists a protocol to have a single source of authentication called oauth [3], which allows third-party apps to grant access to an HTTP service on behalf of the owner of the resource, or by allowing the third-party application



to obtain access on its own behalf. This protocol solves the problem of having different implementations and tokens for authentication within the architecture.

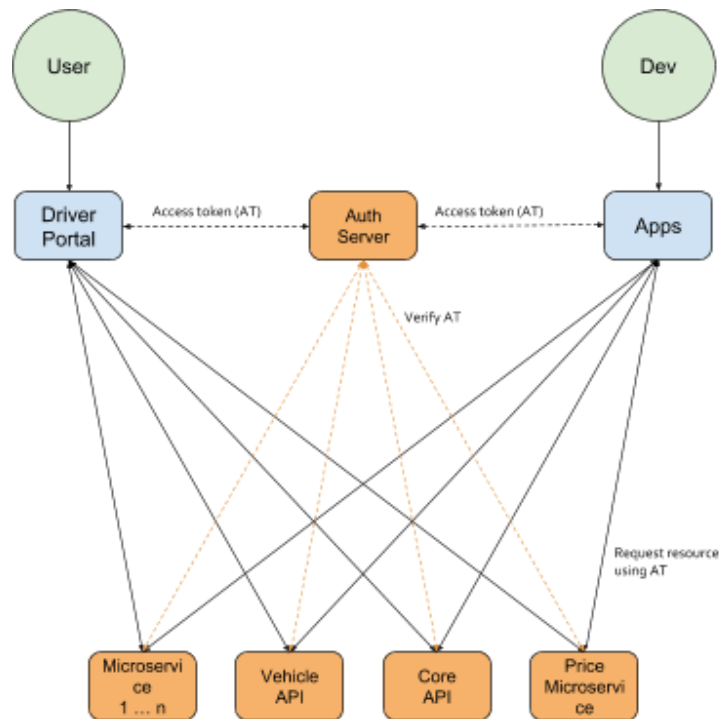


Image 4.4.1.1 - OAuth requests where tokens are verified by Auth Server.

#### 4.4.2. Jwt token format proposal

Although this is a great improvement over the current implementation, it still requires each service to track the state of the users authentication. JSON Web Tokens (JWT) provides a self-contained way of authenticating a user, eliminating the need to query the database more than once. JWT uses a cryptographic signature algorithm to verify user data that is stored in the token payload, this may bring a security concern to the table. If the private key is lost, all requests may be compromised.

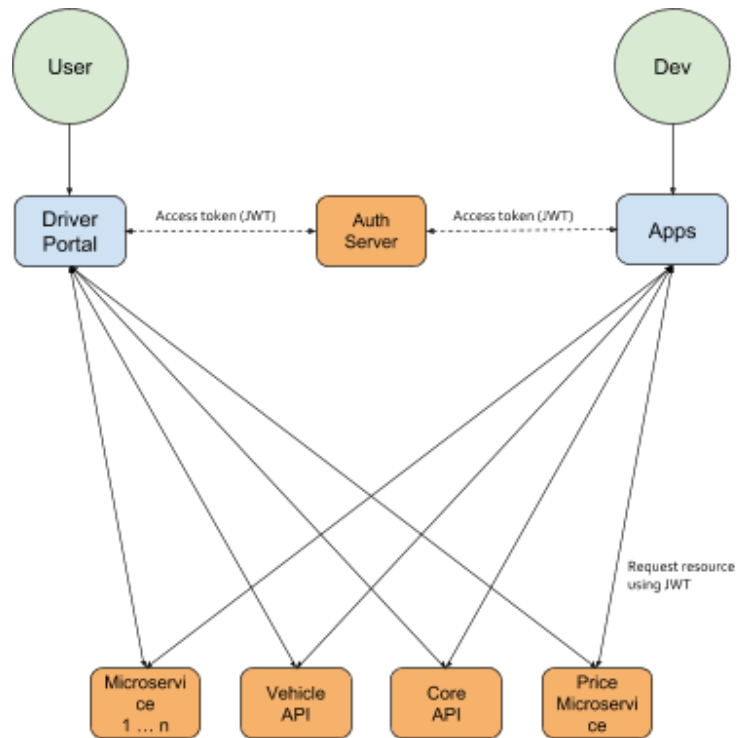


Image 4.4.2.1 -OAuth with stateless JWT token requests.

#### 4.4.3. Proposal API Gateway

Another common structure that allows services to be used by external agents is the API Gateway. It allows for a central middleware in which authentication and authorization is handled, where the microservices are shielded from public access, and all communication is established through the API Gateway [4].

Next to authentication, the gateway could optimize the endpoints so that no multiple requests are needed from external agents to gather different types of resources. These calls could be made internally to the microservices behind the gateway. This also opens the possibility the freely change the microservices without changing the public endpoints exposed by the gateway, and even offers slow or instant transitions to different versions of microservices.

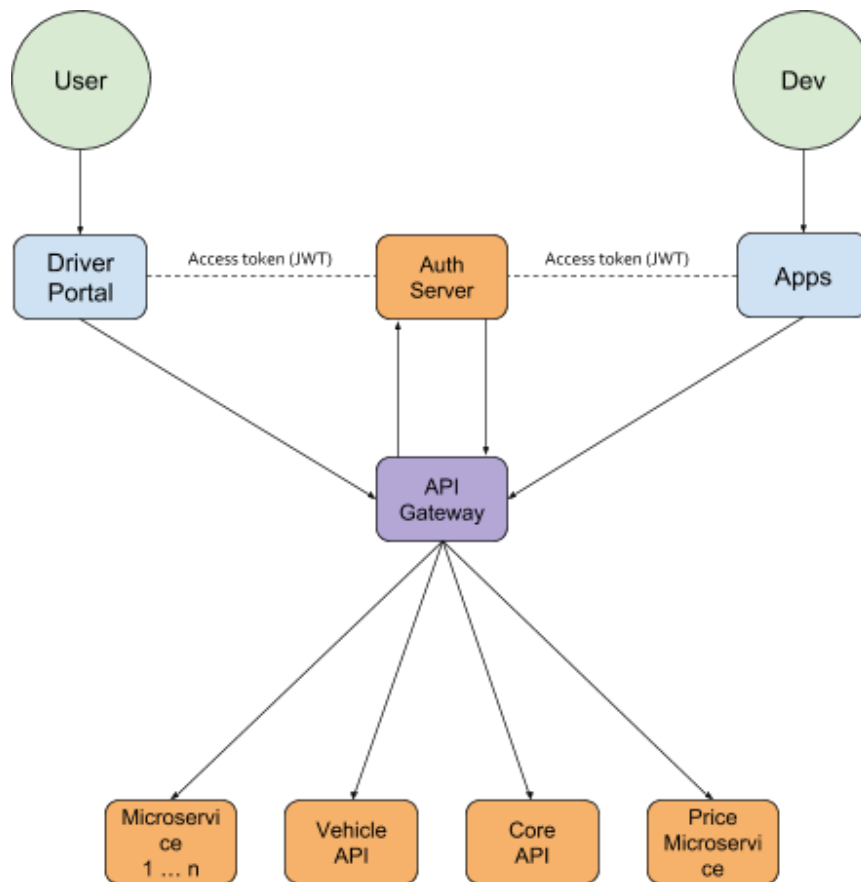


Image 4.4.3.1 - API Gateway.

The different proposals explain the improvements they may bring over some system. But the advice given is not tied to this project, instead to the entire Dispatch API. It's advised to have a constructive dialogue about the future of the company, and the way it's planning to scale. One could put a API Gateway in front of a monolithic app to help with transitioning to a microservice-oriented app.

## 4.5. Database

The database must be capable of determining whether a virtual perimeter contains a set of coordinates, more specifically, it must adhere to The Open Geospatial Consortium (OGC) Simple Feature Access ISO 19125-1 [5] and ISO 19125-2 [6], including spatial data types, analysis functions, measurements and predicates for this requirement, or have some comparable implementation. The scenario presented in image 4.5.1 should be replicable.

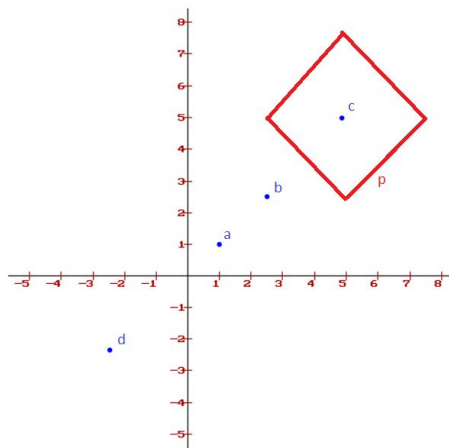


Image 4.5.1 - Four Points and one Polygon p containing Point c.

#### 4.5.1. OpenGIS Compatible databases

MYSQL's innate integrity is a good reason to opt for a full MYSQL database setup. MariaDB is a fork of MYSQL that performs better according to benchmarks, however they don't always translate to real life situations. It's easy to migrate from MYSQL to MariaDB, so choosing MYSQL at first could be preferable as an instance of MYSQL is already used at TaxiID. PostgreSQL offers a spatial database extender for that is OpenGIS compliant called PostGIS that adds support for geographic objects and location queries.

All spatial data types inherit properties such as type and spatial reference identifier (SRID). For rigorous documentation, both PostGIS documentation [7] and MYSQL documentation [8] could be consulted. When a generic geometry column, or point column is created, points can be inserted as shown in snippet 4.5.1.1.

```
START TRANSACTION;
    SET @a = ST_GeomFromText('POINT(1 1)');
    INSERT INTO point (point) VALUES (@a);
    SET @b = ST_GeomFromText('POINT(2.5 2.5)');
    INSERT INTO point (point) VALUES (@b);
    SET @c = ST_GeomFromText('POINT(5 5)');
    INSERT INTO point (point) VALUES (@c);
    SET @d = ST_GeomFromText('POINT(-2.5 -2.5)');
    INSERT INTO point (point) VALUES (@d);
COMMIT;

START TRANSACTION;
    # First and last point must be the same
    SET @a = PolygonFromText('POLYGON((2.5 5,5 7.5,7.5 5,5 2.5,2.5 5))');
    INSERT INTO polygon (polygon) VALUES (@a);
COMMIT;
```

Snippet 4.5.1.1 - Inserting points or polygons in an SQL database.

It is evident that c is contained in p. To determine which points are contained in p, the function as seen in Snippet 4.5.1.2 can be used, which returns the point with coordinates [5, 5] as expected.

<pre>// All points contained in polygon SELECT ST_ASTEXT(POINT) FROM POINT WHERE   ST_CONTAINS(     (       SELECT POLYGON       FROM POLYGON       WHERE id = 1     ),     POINT   );</pre>	<pre>// All polygons containing point SELECT ST_ASTEXT(POLYGON) FROM POLYGON, POINT WHERE   POINT.id = 3 AND ST_CONTAINS(     POLYGON.polygon,     POINT.point   )</pre>
--	--

*Snippet 4.5.1.2 - Find points in polygon, Find polygons containing point.*

#### 4.5.2. OpenGIS Incompatible databases

MongoDB doesn't offer OpenGIS implementations but has geospatial query operators that may provide enough functionalities for current requirements [9]. The argument for choosing one over the other depends on the vast differences between SQL and NoSQL, next to performance and extensiveness of geospatial features. The setup displayed in image 4.5.1 is recreated in MongoDB using queries shown in snippet 4.5.2.1.

```
db.point.insertMany([
  { shape: { type: "Point", coordinates: [1, 1] } },
  { shape: { type: "Point", coordinates: [2.5, 2.5] } },
  { shape: { type: "Point", coordinates: [5, 5] } },
  { shape: { type: "Point", coordinates: [-2.5, -2.5] } },
])

db.polygon.insert({
  shape: {
    type: "Polygon",
    coordinates: [ [ [2.5, 5], [5, 7.5], [7.5, 5], [5, 2.5], [2.5, 5] ] ]
  }
})

db.point.createIndex({ 'shape': '2dsphere' })
db.polygon.createIndex({ 'shape': '2dsphere' })
```

*Snippet 4.5.2.1 - Inserting points or polygons in a NoSQL database.*

<pre>// All points contained in polygon var p = db.polygon.find({})  db.point.find({   shape: {     \$geoWithin: {       \$polygon: [         [2.5, 5],         [5, 7.5],         [7.5, 5],         [5, 2.5],         [2.5, 5]       ]     }   } })</pre>	<pre>// All polygons containing point var p = db.point.findOne({ coordinates: [5, 5] })  db.polygon.find({   shape: {     \$geoIntersects: {       \$geometry: {         type: "Point",         coordinates: [5, 5]       }     }   } })</pre>
---	--

#### *Snippet 4.5.2.2- Find points in polygon, Find polygons containing point.*

Next to database solutions for this requirement, services exist that are capable of geofencing. Although these services may not be free, and the added dependencies restrict extensibility.

#### 4.5.3. Performance and Clustering Trade-offs

Agarwal & Rajan state that NoSQL take advantage of cheap memory and processing power, thereby handling the four V's of big data more effectively, but lack the robustness over SQL databases [10]. The report dives deeper into spatial queries and concludes that their tests suggest that MongoDB performs better by an average factor of 10, which increases exponentially as the data size increases, but lack many spatial functions that OpenGIS supports.

Although improvements have been made [11] after the cited paper Schmid et al. 2015 [12] was published. The team argues that clustering is much easier in MongoDB, which may be important in the future when the company grows. As the required functionalities exist in both SQL and NoSQL, it is beneficial to opt for MongoDB for its performance and alignment with the teams experience. Although if robustness is desired, or extra GIS functionalities required, SQL should be taken into consideration.

## 4.6. API

An important choice that has to be made is the framework in which the project is going to be built. The team has experience with Loopback 3.0 [13], but considering the fact that this microservice is very small, and may not need the large amount of abstractions, Express.js is more suitable for the job. Although this means that required functionalities, that come out of the box with Loopback, have to be replaced.

#### 4.6.1. Required Endpoints

The API should be capable of exposing endpoints (that are going to be specified in more detail in the next phase) that are available to the DriverPortal and to external services. The endpoints for the DriverPortal should expose CRUD operations on resources that are used to calculate a trip. The endpoint for external services has only one task, given some trip information, a price has to be calculated based on the rules of the application that has been used.

#### 4.6.2. Express VS Loopback

As mentioned, the team has experience with Loopback, and having most code written in Loopback, making it easier to transfer pieces of functionality between projects. It has a built in ORM including CRUD endpoints.

On the other hand, Loopback has a steeper learning curve, stagnating velocity among external or new developers. Keeping the code base up to date may be harder because of increased amount of dependencies. There's no clear winner. The best choice should be the result of a consensus between core developers.

## 4.7. Database Schema

When a user being tied to a application is authenticated, prices can be calculated depending on various variables. Some variables should be passed each call, like the destination, departure location, timestamps and other important information. Some information will not change each ride, this should be defined and could be changed by the user, and should be stored in the database of the Price API.

#### 4.7.1. Relational Database

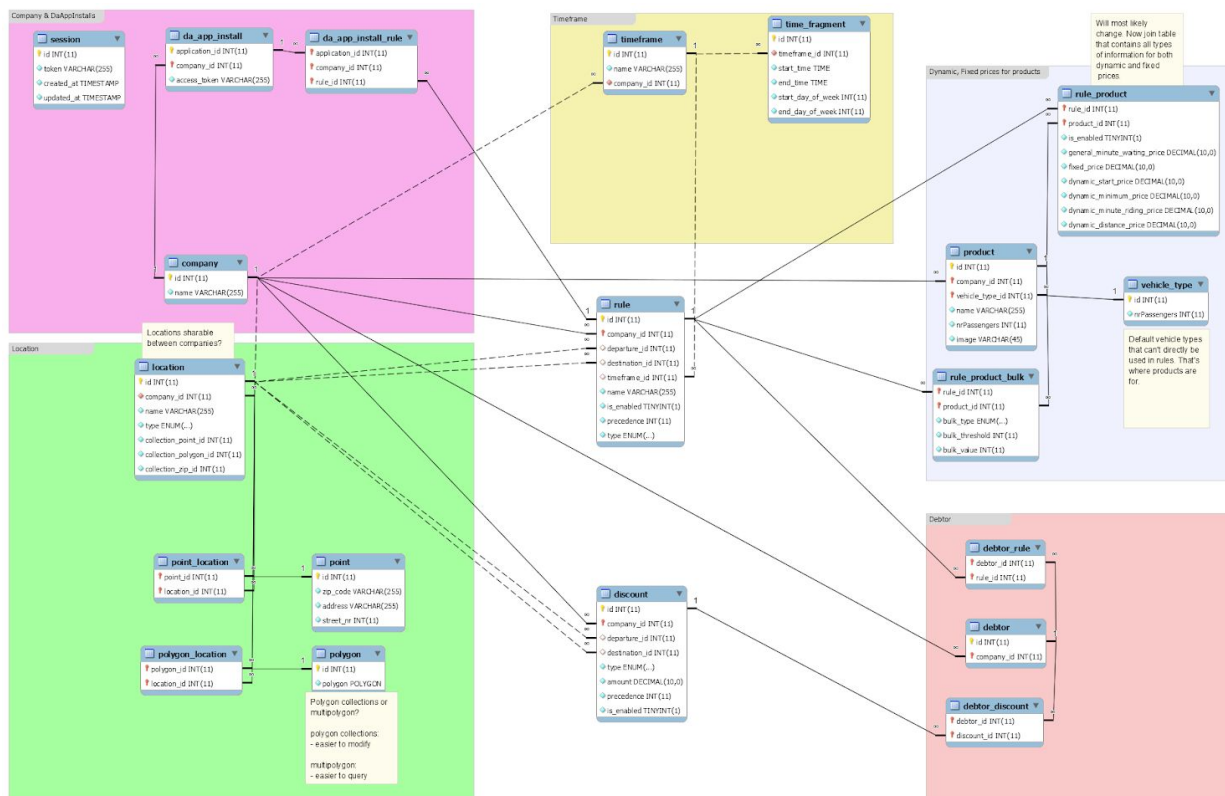


Image 4.7.1.1 - Rough schema for a relational database.

This schema cannot represent a NoSQL database, where relations are embedded. But the general idea in this schema could still be used and translated to NoSQL. The MongoDB documentation communicates schema information by presenting a document diagram. The main differences between relational and non-relational databases have to be taken into account, embedding and referencing over.

#### 4.7.2. Non-Relational Database

```
// Application document
// with embedded settings
{
  _id: <ObjectId>,
  user_token: "",
  settings: {
    is_begin_end_same_address: true
  }
}
```

```
// Rule document
// with embedded rule type
// with references to one discount (or many)
{
  _id: <ObjectId>,
  application_id: <ObjectId>,
  created_at: ISODate("2013-10-02T01:11:18.965Z"),
  updated_at: ISODate("2013-10-02T01:11:18.965Z"),
  is_enabled: true,
  type: "dynamic",
  rule_settings: {
    minimum_price: ... ,
    start_price: ... ,
    ...
  },
  discount: {
    ...
  }
}
```

*Snippet 4.7.2.1 - Difference relational and non-relational database.*

## 4.8. User Interface

Following the principles Shneiderman's mantra, the user should be able to have an overview of the data, then be able to zoom and filter, then get details on demand [14]. The dashboard displays the most crucial components in which items (rules, discounts, vehicle types, e.g.) cannot be edited, but can be enabled or disabled. The settings panel may contain inputs that are allowed to mutate information because the settings are seen as a single item. The rules table should visualize to the user in which order the rules fire (either the order of rows, or a specific column with ordering numbers).

Clicking on a row in one of the tables brings the user to the corresponding detail page: rules, discounts, or vehicles. In each detail page, the rule can be mutated in a most flexible way. The rules detail page contains all the information linked to one single rule. The rule has one type but many options. Each option adds more information to the rule, but some options should be constrained. For example, defining two start prices should not be possible, but defining two bulk price thresholds should be.

## 4.9. Continuous Integration, Continuous Deployment & Testing

Depending on the way a project is set up, different CI providers offer better choices over others. This chapter will only dive into the subject shallowly, because TaxiID has already adopted BuddyWorks.

	Jenkins	Travis	Circle	BuddyWorks
Team preference				✓
Free	✓	public repo	✓	✓ max 5 projects



Cloud-based		✓	✓	✓
GUI pipeline-builder				✓
SSH	✓ local		✓	Indirect through predefined script
Metadata collection	✓ local		✓	coverage report gives 404

*Table 4.9.1 - Comparison between CI providers.*

## 4.10. Testing

Software Reliability is defined as the probability of an item to perform a required function under stated conditions for a specified period of time. New features often introduce bugs by adding functionalities that are broken, although the reliability of the existing functionalities may also be impacted because of changes in the existing code. To prevent units of code from malfunctioning, regression tests may be implemented to validate whether a unit still functions according to a set of conditions.

Static and dynamic tests may be performed using the framework Mocha [15] and the assertion library Chai [16].

On top of that, Microsoft's new language Typescript could be used to replace EcmaScript, enabling type checking during development, boosting development velocity in the long run by preventing type related bugs from being introduced.

## 5. Conclusion

### 5.1. Frontend

The first non-functional requirement states that the solution should be seamlessly integrated in the portal. On top of that, a user shouldn't have to log in again to make use of the pricing service from within that portal. Iframes, objects and embeds have been mentioned as potential solutions to integrate a frontend in several distinct portals. This problem affects more than just the pricing project, therefore a decision must be made on a higher level before the frontend will be integrated, but the decision is not required for the first sprint to start. The options that are available are: an integrated view inside the existing DispatchAPI project or a separate solution built in Vue2 with a material design style that can be integrated using an iframe.

### 5.2. Backend

The backend should be loosely coupled, but should be accessible by all users who are able to authenticate and authorize themselves. It's advised to implement the system as a microservice, because it separates the concern effectively. By implementing the system as a module, the implementation is entirely dependent on the existing system it's implemented in, stalling modernization of architecture in the long run. The solution that is presented in the pregame solves this challenge by having one microservice handle the requests that are in some cases routed through the DispatchAPI. The requests sent by a user from any portal should be directed at the microservice, while price calculation requests should be routed through the DispatchAPI. Loopback should be used as a framework, preferably in combination with typescript.

### 5.3. Functionalities

The core functionality of the system is to calculate a price based on rules defined by the user. The user is able to define which Dispatch API application installations (DaAppInstallations) may use these rules, but also which debtors may use these rules. If a ride is booked by the passenger, the passenger may be entitled to a discount if he or she orders the ride while being related to a debtor that is linked to a discount, or if the company has discounts that are matched with the ride. In this case other rules may apply. In any other case, the rules that are tied to the DaAppInstallation from which a ride is booked are used.

The other main functionality encapsulates all the steps that a user must take to set up the prices for the company. By generalizing concepts such as time and place as much as possible, the user can reason about his decisions more easily. For example, a location can be defined as a collection of zip codes, a collection of points or a collection of area's. To be more concrete, a user may define a location named 'Falke Hotels', using a list of zip codes. Next the user draws an area on top of Schiphol to define another location. Now these locations may be used in a rule that defines fixed prices from Falke Hotels to Schiphol. The user selects the price, the start location and end location he has just defined. The user also wants to give passengers that have a relation with the Falke debtor have a 10% discount on fridays. The user creates a discount, fills in 10% discount and adds a timeframe within which this discount is applicable. The user selects 'add timeframe', and selects the hours of the week in a timeframe view. He selects all the hours on friday and names this timeframe 'fridays'. The user connects the rule and the discount to a debtor name 'Falke', now all the passengers will pay fixed prices from hotels to Schiphol with a 10% discount on friday.

A passenger who books a ride from a Falke hotel requests the price, as he's tied to a debtor, he sends a debtor identifier to the system. The API selects the rules that are tied to the debtor (if no rules are tied, the system will fall back on rules defined for the DaAppInstallation) within the company. The API tries to find a departure location that matches with a rule. But the passenger travels to amsterdam, not to Schiphol, therefore no rule was found. The API finds a dynamic pricing rule, so the price is calculated using a start price, price per kilometer and price per minute. The passenger has ordered an electric limousine (defined as a custom vehicle type by the user), so the most expensive tariffs are used. The passenger also lets the limousine wait for 10 minutes, so the price goes up a bit. Because it's friday, the passenger is lucky to have a 10% discount and passes a bulk threshold at 30 kilometers traveled, lowering the price per kilometer from that point onward. As the electric limousine reaches the location in Amsterdam, the driver adds a small additional fee on top of the calculated price because the passenger spilled a drink inside the limousine, which is handled outside of the price calculation.

All the steps demonstrated in the story can be handled by the proposed system functionalities and data structure as explained in the Phase I - Pregame document. Some edge cases like layered area's are resolved by defining precedences on rules and discounts. The edge case of having a neighbour profit from hotel discounts, is by having rules and discounts be tied to debtors. The edge case of having to define many hotels by drawing area's around them on a map can be handled by defining specific points instead. The edge case of no rules being found is resolved by returning an error, this may be subject to change.

## 5.4. Authentication and Authorization

When speaking about microservices, authentication is the immediate next concern. If requests can be sent to the microservice directly, there must be a solution implemented to authenticate and authorize the user autonomously. As with the frontend discussion, this matter is of importance if more microservices are implemented in the future. It may be beneficial to introduce a single solution of authentication and authorization. This is suggested in the document by implementing an authentication server that provides a token that can be validated at a microservice level. If this is not desired, a similar authentication flow can be implemented as described by Marco as used in current systems.

## 5.5. Database

MongoDB should be used over an SQL database because of its scalability. MongoDB supports geographical location types, geospatial queries including the predicate to check which polygons contain a single point, or retrieving all points contained within a single polygon.

## 5.6. User Interface

The user interface will contain an overview showing the main concepts that a user has to maintain: pricing rules, locations, discounts. The UI should be focussed on linear navigation with overviews of detail pages. The UI will contain a screen to assign rules and discounts to DaAppInstallations and debtors, a screen to define locations, a screen to edit rules, a screen to modify vehicle types, and a screen to define timeframes.

## 6. References

- [1] "ST\_Contains." [Online]. Available: [https://postgis.net/docs/ST\\_Contains.html](https://postgis.net/docs/ST_Contains.html). [Accessed: 06-Feb-2018].
- [2] "MySQL :: MySQL 5.7 Reference Manual :: 12.15.9.1 Spatial Relation Functions That Use Object Shapes." [Online]. Available: [https://dev.mysql.com/doc/refman/5.7/en/spatial-relation-functions-object-shapes.html#function\\_st-contains](https://dev.mysql.com/doc/refman/5.7/en/spatial-relation-functions-object-shapes.html#function_st-contains). [Accessed: 07-Feb-2018].
- [3] "OAuth 2.0 — OAuth." [Online]. Available: <https://oauth.net/2/>. [Accessed: 09-Feb-2018].
- [4] "Why Use an API Gateway in Your Microservices Architecture?," *NGINX*, 19-Apr-2017. [Online]. Available: <https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/>. [Accessed: 15-Feb-2018].
- [5] "Simple Feature Access - Part 1: Common Architecture | OGC." [Online]. Available: <http://www.opengeospatial.org/standards/sfa>. [Accessed: 07-Feb-2018].
- [6] "Simple Feature Access - Part 2: SQL Option | OGC." [Online]. Available: <http://www.opengeospatial.org/standards/sfs>. [Accessed: 07-Feb-2018].
- [7] "Chapter 2.4. Using PostGIS: Data Management and Queries." [Online]. Available: [https://postgis.net/docs/manual-2.4/using\\_postgis\\_dbmanagement.html#PostGIS\\_GeographyVS\\_Geometry](https://postgis.net/docs/manual-2.4/using_postgis_dbmanagement.html#PostGIS_GeographyVS_Geometry). [Accessed: 07-Feb-2018].
- [8] "MySQL :: MySQL 5.7 Reference Manual :: 11.5.2.2 Geometry Class." [Online]. Available: <https://dev.mysql.com/doc/refman/5.7/en/gis-class-geometry.html>. [Accessed: 07-Feb-2018].
- [9] "Geospatial Query Operators — MongoDB Manual 3.6." [Online]. Available: <https://docs.mongodb.com/manual/reference/operator/query-geospatial/>. [Accessed: 07-Feb-2018].
- [10] K. S. R. Sarthak Agarwal, "Analyzing the performance of NoSQL vs. SQL databases for Spatial and Aggregate queries," International Institute of Information Technology Hyderabad Gachibowli, India, Sep. 2017.
- [11] "Geospatial Performance Improvements in MongoDB 3.2," *MongoDB*. [Online]. Available: <https://www.mongodb.com/blog/post/geospatial-performance-improvements-in-mongodb-3-2>. [Accessed: 12-Feb-2018].
- [12] Stephan Schmid Eszter Galicz, "Performance investigation of selected SQL and NoSQL databases," Bundeswehr University Munich, June 9-12, 2015.
- [13] "LoopBack 3.x | LoopBack Documentation." [Online]. Available: <https://loopback.io/doc/en/lb3/>. [Accessed: 12-Feb-2018].
- [14] C. T. Architecture, "Shneiderman's mantra - Coding the Architecture." [Online]. Available: [http://www.codingthearchitecture.com/2015/01/08/shneidermans\\_mantra.html](http://www.codingthearchitecture.com/2015/01/08/shneidermans_mantra.html). [Accessed: 15-Feb-2018].
- [15] "Mocha - the fun, simple, flexible JavaScript test framework." [Online]. Available: <https://mochajs.org/>. [Accessed: 21-Feb-2018].
- [16] "Chai." [Online]. Available: <http://chaijs.com/>. [Accessed: 21-Feb-2018].

# **Appendix B**

## **Sprint Summaries**

### **Sprint Summaries**

<https://goo.gl/BR75HN>

