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

June 2018

A rule-based geospatial reasoning system for trip price calculations

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

Acknowledgements

I would like to express my gratitude to Willem Brouwer for helping me leverage the quality of this thesis.

I would also like to thank the people at taxiID for granting me the responsibility of creating their trip price calculation system independently.

Finally I would like to thank my parents for their limitless support.

Stefan Schenk

Andijk, 01-05-2018

Abstract

kks32: Abstract, will be the last step

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 benifits 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

1 Introduction **1**

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

2 Encoding Locations **7**

2.1 Introduction 7
2.2 A Brief History Of Geographic Locations 7
2.3 Requisite Location Types 8
2.3.1 The Point 8
2.3.2 The Area 9
2.3.3 Postal Codes, Addresses, and Polygons 10
2.3.4 Requirements for Location Matching 11
2.4 Literature Review 11
2.5 Database Prerequisites 12
2.5.1 OpenGIS Compatible databases 13
2.5.2 OpenGIS Incompatible databases 14
2.6 Overlapping Locations 17
2.7 Conclusion on Encoding Locations 17

3 System Architecture **19**

3.1 Introduction 19
3.2 Architectural Patterns 19
3.2.1 Monoliths 20
3.2.2 Microservices 21

3.2.3 Frontend and Backend 21
3.3 Information Dependencies 22
3.4 Authentication and Authorization 23
3.4.1 OAuth 2.0 23
3.4.2 JSON Web Token 24
3.4.3 API Gateway 25
3.5 Methods and Techniques 26
3.5.1 Backend Framework 27
3.5.2 Frontend Framework 27
3.5.3 Database 28
3.6 Conclusion on System Architecture 29

4 Trip Price Calculation System **31**
4.1 Introduction 31
4.2 The System Structure 31
4.3 Matching Criteria 33
4.3.1 Locations 33
4.3.2 Timeframes 35
4.4 The Trip Price Calculation 37
4.4.1 Incoming Request 38
4.4.2 Data Aggregation 39
4.4.3 Calculation 40
4.5 Price Calculation Types 42
4.5.1 Dynamic 43
4.5.2 Fixed 44
4.5.3 Meter 44
4.6 Threshold Calculations 44
4.7 Determinism 46
4.8 Breakdown 47
4.9 Conclusion on Trip Price Calculation System 49

5 The Portal **51**
5.1 Introduction 51
5.2 Visual Hierarchy 51
5.2.1 Essential Components 51
5.2.2 Expressing Order 52
5.3 Design 53

5.4 Products . . . 54
5.5 Pricing . . . 54
5.5.1 Discounts . . . 54
5.5.2 Rules . . . 55
5.5.3 Priority . . . 55
5.6 Locations . . . 55
5.6.1 Point . . . 56
5.6.2 Area . . . 57
5.7 Apps . . . 58
5.8 Conclusion on The Portal . . . 58

6 Realization **59**
6.1 Introduction . . . 59
6.2 Sprint 1 - Dynamic Price Calculations . . . 59
6.3 Sprint 2 - Authentication and Authorization . . . 59
6.4 Sprint 3 - Products and Pricing . . . 60
6.5 Sprint 4 - Apps and Timeframes . . . 60
6.6 Sprint 5 - Thresholds . . . 61
6.7 Sprint 6 - Locations . . . 61
6.8 Sprint 7 - Subrules . . . 62
6.9 Sprint 8 - Polishing . . . 62

7 Conclusion **63**

References **65**

List of figures **67**

List of tables **69**

Appendix A Pregame **71**

Appendix B Sprint Review and Proposal Slides **107**
B.1 Sprint 1 - Dynamic Price Calculations . . . 107
B.2 Sprint 2 - Breakdown Proposal . . . 114
B.3 Sprint 2 - Authentication Proposal . . . 120
B.4 Sprint 2 - Authentication and Authorization . . . 125
B.5 Sprint 3 - Products and Pricing . . . 131

B.6 Sprint 4 - Apps and Timeframes 136
B.7 Sprint 5 - Thresholds 143
B.8 Sprint 6 - Locations 148
B.9 Sprint 7 - Subrules 152
B.10 Sprint 8 - Polishing 156

Chapter 1

Introduction

What was once an ordinary startup known as Uber, is now the most famous taxi dispatch company in the world [1]. A similar startup was founded in the same year, called taxiID; an Amsterdam based company providing end-to-end cloud solutions and mobile applications for taxi companies. Hailing a taxi has rarely been performed by sticking out ones hand, hoping to catch the attention of a bypassing taxi driver ever since. The ability to order a cab lies at everyone's fingertips, literally. 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 how this issue is translated into the assignment.

1.1 Context

taxiID was founded as a startup that successfully introduced smartphone taxi booking in The Netherlands, offering a wide variety 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 when developing applications that depend on counties' infrastructure and postal code systems. The 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

The new YDA apps depend on the price calculation module that is part of the legacy system, for which it was once designed and implemented. Taxi companies have prices defined for the vehicle rides that they offer to passengers. If YDA clients want to price their products, they are obligated to use the legacy taxiID portal, which has to store company information in a platform that is not related to YDA. This makes little sense, as much as it is efficient from a technical point of view, as well as being easy to maintain and extend. The legacy price calculation module knows three types of pricing rules: fixed prices based on postal codes or addresses, tier prices based on kilometer thresholds, and dynamic calculations based on the distance and duration of a ride. These rules will be selected depending on the characteristics of a taxi trip, but they will be matched in the same order as mentioned. A company may have as many pricing rules as required, only one rule will be used to calculate the final price. 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	...

Table 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 and/or addresses, amount of passengers, and desired vehicle types that are sent by the booking app with the fixed pricing rules. A 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 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 4. 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 again, 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 major issues if the error passes through deployment 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 monolithic, highly coupled system, as it is today. Clients must be able to set up pricing rules and discounts through the YDA portal. It is important that the feature that allows clients to define locations for the pricing rules, are usable in countries that have no workable postal code system.

1.4 Research

Four 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, while keeping the locations conveyable. An alternative technique to comparing and storing locations must be explored, that does not regress in comparison to the old technique, that uses CSV files and postal codes. Edge cases must be dealt with to prevent a situation in which users are unable to achieve that which they could with the old system. The best way of integrating the new system in the existing system architecture should be investigated. But improvements must not be withheld, proposals with good arguments should be presented with regards to the chosen technologies, authentication, authorization, and system design. The price calculation algorithm must be examined, including the logic of matching rules. The best way to communicate this logic to the user through the YDA portal in a way that makes reasoning

about the price definitions possible, must be found. All the functionalities mentioned must be usable anywhere in the world.

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. Which location encoding is sufficient for this system to be operational?
    1.1. How can legacy location definitions be improved to be universally interpretable?
    1.2. In what way can location matching be improved?
    1.3. Which Database Management Systems are candidate for handling this project's use cases?
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 system architecture?
    2.2. How is state shared and synchronized between system components?
    2.3. What is the most applicable authentication method?
3. Which logic and data is required in the backend to reliably calculate a trip price?
    3.1. Which criteria should regulate whether rules match?
    3.2. How can determinism of price computations be guaranteed?
    3.3. In what way can the three original pricing rule types be implemented? (fixed, dynamic, and threshold prices)
4. Is it possible to communicate the inner workings of the system through the user interface?
    4.1. Which backend concepts are essential to display in the frontend?

4.2. Which design practices allow users to understand coherence of different elements that make up a rule?
4.3. How should a user know what the outcome of his interactions with the system are?

Answering these questions will lead to the implementation of a solid, straightforward, user-friendly system that utilizes the user interface to communicate the inner-workings of the rule-based price calculation system.

1.5 Process

A desire from within taxiID to use the SCRUM methodology to potentially improve their development process is an important factor to set up this project in a way that would introduce the team to SCRUM without forcing developers and CEO's to adopt it right away. All team members are familiarized with tools, roles, workflows, and the project artifacts somewhat indirectly. Because of the novelty of SCRUM in regard to the product owner, a pregame phase is introduced for preparation purposes, see Table 1.2. A written working method is provided to the product owner, see Appendix A, Phase I - Pregame. The interpretation of the product owners product vision and the reflection from a developer viewpoint is documented, so that miscommunications and misinterpretations can be resolved before the project is started. It contains an architectural vision and a proposed solution, which is agreed upon by the product owner before the backlog is created. Reading the document is recommended if more knowledge about the process and context of the assignment is desired.

Phase I - Pregame			Phase II - Game							
week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11
product definition	architectural vision	proposed solution	sprint 1	sprint 2	sprint 3	sprint 4	sprint 5	sprint 6	sprint 7	sprint 8

Table 1.2 Project roadmap

Chapter 2

Encoding Locations

2.1 Introduction

The term 'geospatial' denotes "relating to the relative position of things on earth's surface". This chapter concretizes the definition of a location. The way of storing and matching locations, and solving the complementary problems will be discussed.

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 ϕ, in conjunction with a chronometer to determine the longitude λ of a location on the earth's surface. The combination of coordinates is a distinct encoding of a location.

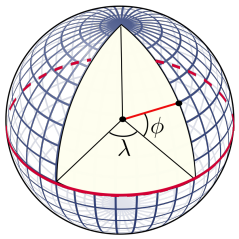


Fig. 2.1 A perspective view of the Earth showing how latitude and longitude are defined on a spherical model.

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, and the new Galileo global navigation satellite system, provide tracking

with a precision down to the meter 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, and 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. This forces the legacy system to support addresses for the fixed pricing functionality as well. 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 radiuses and shapes. You could prepend 'the location of' to the following terms as an example: America, the birthplace of Socrates, 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 Requisite 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 section 3.2.1 of Appendix A, an agreement was made on what the terms "area" and "point" meant. The MySQL documentation notes that "The term most commonly used is geometry, defined as a point or an aggregate of points representing anything in the world that has a location." in [2]. During the process of implementing TPS, the definitions of a location have been refined to represent a common and useful understanding.

2.3.1 The Point

A point is a unique place expressed as a distinct coordinate pair. An address in the legacy system could be translated to a point. For example, the address that is tied to Schiphol arrival is: Aankomstpassage, 1118 AX Schiphol Centrum. The point that encodes this location is (52.308891, 4.760900). This location is contained in the set of all possible points on Earth, which could be expressed using set builder notation:

$$P = \{(\phi, \lambda) \in \mathbb{R}^2 | -90 < \phi < 90, -180 < \lambda < 180\}$$

$$(52.308891, 4.760900) \in P$$

A point itself can not be used to match whether another point is contained within it, because the probability of a match is infinitesimal. Only when decimals were disregarded to decrease the precision of a point, or if the origin of the point would be provided by some service, the distinct point would be a viable option to match locations.

2.3.2 The Area

An area is a set of points points with an infinite granularity. This definition allows for an area to have holes inside them, consist of other locations and contain other locations, and be infinitely precise. The most useful property of this area is to check whether a point is contained within the area, or which areas contain a given point. For this to be the case, the points must be packed together to form a shape. This definition, however conceptually valuable, will not be of much practical use. For example, P is an infinitely long set of coordinates, an area that represents the earths surface. If ϕ ranged between 0 and 90, the set should describe all points located in the northern hemisphere, but would still be infinitely long. Checking whether a given point is contained by checking an infinite amount of real number pairs will take an infinite amount of time in the worst case scenario. Such an area can be described as a subset of all points:

$$a_1 \subseteq P$$

The set of all possible areas can be defined by the power set of P:

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

such that an arbitrary subset of points, called an area, is an element of all possible areas A:

$$a \in A$$

At the equator, 1 degree is 111320m, so 0.000001 degrees is around 11cm. Six decimal places will be sufficient for location matching for this application. But even when reducing coordinates to having six decimal places, it would be impractical. For this reason, it is more realistic to only describe the rough edges of an area using a polygon shape. Polygon geometry is widely supported by database systems. Instead of checking for a single point in a non-terminating iteration over all points in an area, a mathematical calculations could be used to check whether a unique point is contained within the polygon.

2.3.3 Postal Codes, Addresses, and Polygons

All postal codes that start with a ten describe the city of Amsterdam, the entire area of Amsterdam can be drawn as a big polygon containing all the postal codes that start with a ten.

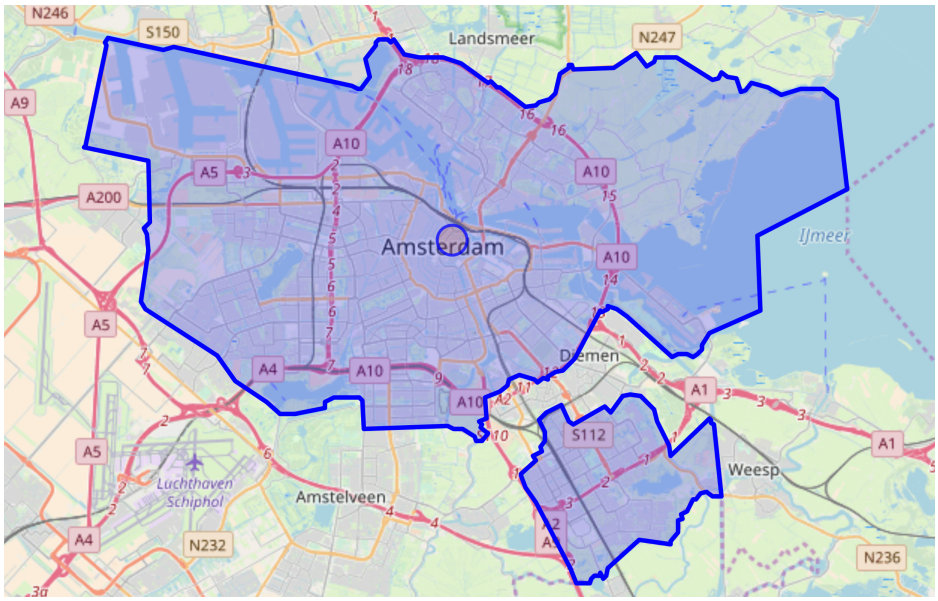


Fig. 2.2 Amsterdam - A single area comprised of multiple locations.

In reverse, this procedure would not work. If a polygon was drawn cutting Amsterdam in half diagonally, a single postal code pattern would never be flexible or precise enough to be able to describe the boundaries of the polygon. One big taxi company making use of taxiID's legacy system is located in the United Arab Emirates. This company would not be able to convert anything at all, because the United Arab Emirates does not have a postal code system to begin with. Regardless, addresses and postal code systems do not provide universally interpretable and precise encodings of locations, especially for the locations that matter for this project: points and areas. They can be ambiguous, imprecise, and nonuniform. In the United Arab Emirates, addresses can still be utilized, even though it is harder to ensure that two addresses match. Street numbers, punctuation, formats and special characters, may cause the matching process to fail. In contrast, polygons would provide unique and precise location definition that is uniform and universal. When moving to other encoding techniques, this usefulness must be preserved.

2.3.4 Requirements for Location Matching

If the following statements are true for a given location encoding using the definitions of the Point and Area, the location encoding is useful and able to operate independently from the postal code and address systems.

Nr	Description
1.	Every location is stored in a database as a single entity
2.	Locations can consist of multiple locations (see figure 2.2)
3.	A predicate of whether a location is fully contained within a location is achievable
4.	A method of finding all locations containing a single location can be used
5.	A method of determining precedence of location in case of overlap must always yield one result, and discard all others
6.	Locations must be importable from external sources

Table 2.1 Location matching requirements.

2.4 Literature Review

In [3], CEO Chris Cheldrick explains how locations can be communicated more effectively by describing a three by three meter areas using three words that are assigned to the area. The system aims to solve the problem of ambiguity in address or postal code systems. The what3words API offers functionalities that can find what3word geocodings near a specified latitude and longitude location. The system is able to find results within a clamped area, as documented in [4], effectively acting like a spherical circle with a given radius in which points can be contained. In the paper [5] Markus elaborates on the distinction between structure-based spatial data and point sets, stating that: "Structure-based spatial data types have prevailed and form the basis of a large number of data models and query languages for spatial data". He elaborates on distinctions of operations and predicates between different spatial data models in [6]. Operations such as point-in-polygon test and intersection are categorized as spatial modeling. Regular spatial database systems support a basic Geometry hierarchy of Points, Polygons, MultiPoint and MultiPolygon Classes, as described in the OGC [7] and ISO 19125 [8] standard. MySQL, PostgreSQL, MariaDB and other systems having distinct implementations adhere to the OGC standard. Some other databases like MongoDB adopt

the GeoJSON standard [9], providing similar operations and data types. Xiang et al proposed conventional flattened R-Tree indexing for the less mature MongoDB spatial system [10]. The built in Geohashing method is typically used to index points and centroids, having the possibility to inaccuracies and missing data. Locations should be importable in geography formats. Holmberg extracts data from OpenStreetMap as shape files, see [11, Chapter 6]. He uses two sources: https://www.openstreetmap.org and http://download.geofabrik.de see [11, Chapter 7.3]. The latter is used to obtain data for whole countries. OpenStreetMap offers a downloadable dataset at https://planet.openstreetmap.org from which geographic data can be exported. The OSM Nominatim Usage policy states that no heavy usage is allowed, that bulk geocoding is restricted, that auto-complete is not supported, and that attribution must be displayed [12].

2.5 Database Prerequisites

The database that is used must be able to aggregate all polygons containing a given point. Conversely, it must be able to aggregate all points that are contained within a given polygon. The scenario presented in image 2.3 should at least be replicable.

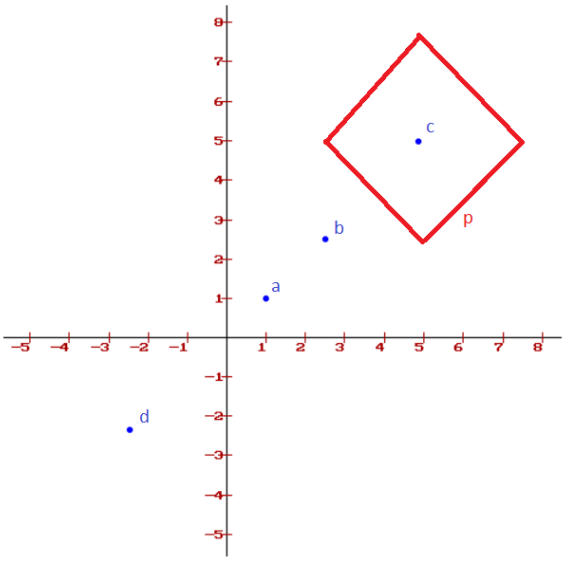


Fig. 2.3 Four Points, one Polygon p containing Point c.

This overly simple example provides proof that a minimal requirement is satisfied, so that a list of candidate Database Management Systems could be constructed. More complex tests have been conducted that involved different shapes, but the desired outcome remains

the same. In all cases, a polygon is a list of coordinates that define a closed path, meaning that the first and last index contain identical points.

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. All spatial data types inherit properties such as type and spatial reference identifier (SRID). For rigorous documentation, both PostGIS documentation [13] and MYSQL documentation [14] could be consulted. When a generic geometry column, or point column is created, points can be inserted as shown in snippet 2.1.

```
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 (@a);
# also insert @b, @c, and @d
COMMIT;

START TRANSACTION;
# First and last point should 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;
```

Listing 2.1 Insert four points and one polygon in MySQL.

It is evident that c is contained in p. To determine which points are contained in p, the function as seen in Snippet 2.2 can be used, which returns the point with coordinates $[5,5]$ as expected.

```
// All points contained in polygon
SELECT ST_ASTEXT(POINT)
FROM POINT
WHERE
ST_CONTAINS(
  (
    SELECT POLYGON
    FROM POLYGON
    WHERE id = 1
  ),
  POINT
)

// All polygons containing point
SELECT ST_ASTEXT(POLYGON)
FROM POLYGON, POINT
WHERE
  POINT.id = 3 AND ST_CONTAINS(
    POLYGON.polygon,
    POINT.point
  )
```

Listing 2.2 Select points contained in polygon, and all polygons containing a point in MySQL.

A multipolygon can be inserted using triple braces, indicating a collection of polygons to be inserted as seen in Figure 2.3. The MultiPolygon class is able to support multiple polygons to be stored as a single entity. The standard provides containment predicate, and methods to distinguish larger locations from smaller ones, which could be used in precedence checks.

```
START TRANSACTION;
# First and last point should be the same
SET @a = GeomFromText('MULTIPOLYGON(((1 1,2 2,3 3,1 1)),((5 5,6 6,8 8,5 5))
  )');
INSERT INTO multipolygon (multipolygon) VALUES (@a);
COMMIT;
```

Listing 2.3 Insert one multipolygon in MySQL.

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 [15]. 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 2.3 is recreated in MongoDB using queries shown in snippets 2.4 and 2.5. Geometry datatypes can be inserted as objects having a type and coordinates property. A polygon can be inserted in the same manner, having multiple points as a list instead of a single point.

```
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' })
```

Listing 2.4 Insert four points and one polygon in MongoDB.

A method named $*geoWithin* can be used to return points that are contained within the polygon. Conversely, all polygons that contain a certain point can be queried using the $*geoIntersects* method as seen in 2.5.

```
// 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]
      ]
    }
  }
})

// All polygons containing point
var p = db.point.findOne({ coordinates: [5, 5] })

db.polygon.find({
  shape: {
    $geoIntersects: {
      $geometry: {
        type: "Point",
        coordinates: [5, 5]
      }
    }
  }
})
```

Listing 2.5 Select points contained in polygon, and all polygons containing a point in MongoDB.

In MongoDB, a multipolygon can be inserted using extra pairs of braces, as shown in 2.6. Any predicate will fail if the type is defined as 'Polygon', but a MultiPolygon is stored in the coordinates property or vice versa. Therefore, it is important to manage the type property as more polygons are to be stored at once.

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

Listing 2.6 Insert one multipolygon in MongoDB.

2.6 Overlapping Locations

If the destination is contained in several polygons associated with multiple rules, which rule should then be used to calculate the final price? A database will just pick the first result when the results are limited to one. Several solutions have been proposed to solve this problem:

1. Using the location with the shortest distance from its centroid to the destination.
2. Picking the location with the smallest area.
3. Picking the location that has the rule with the lowest price.
4. Picking the rule that as the highest precedence assigned by the user.
5. A combination of these proposals.

All listed solutions will work in the databases listed in this chapter, as the centroid and area can be calculated in OGC and GeoJSON databases.

2.7 Conclusion on Encoding Locations

"Which location encoding is sufficient for this system to be operational?"

Addresses and postal codes can be translated to geometric datatypes such as Points, Polygons and MultiPolygons. Geometry based locations can be visualized, and thus interpreted regardless of the country in which a location resides. Matching is done through the OpenGIS or GeoJSON API, by writing geospatial queries depending on the selected candidate database system. Selected candidate database systems are systems that adhere to the OpenGIS or GeoJSON standard, yielding many possibilities, of which MySQL and MongoDB have been proven to be workable. The problem of overlapping locations, can be solved using complex

and straight forward approaches, thus proving this location encoding to be sufficient for this system to be operational.

Chapter 3

System Architecture

3.1 Introduction

The term ’system’ denotes "a set of things working together as parts of a mechanism or an interconnecting network". The family of systems that has formed through preceding architectural design decisions, must be able to integrate the new TPS system. Flows of information are to be aligned with adjacent system components so that dependencies are satisfied, while making use of the most fitting technologies for great adaptation. Existing conventions, methods and styles throughout the technical and conceptual spectrums are applied, enabling the system architecture to evolve consistently at one pace. Additionally, adjacent systems are improved by solutions introduced in this chapter.

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 bigger and smaller shapes in the Figure represent large API’s and smaller services respectively.

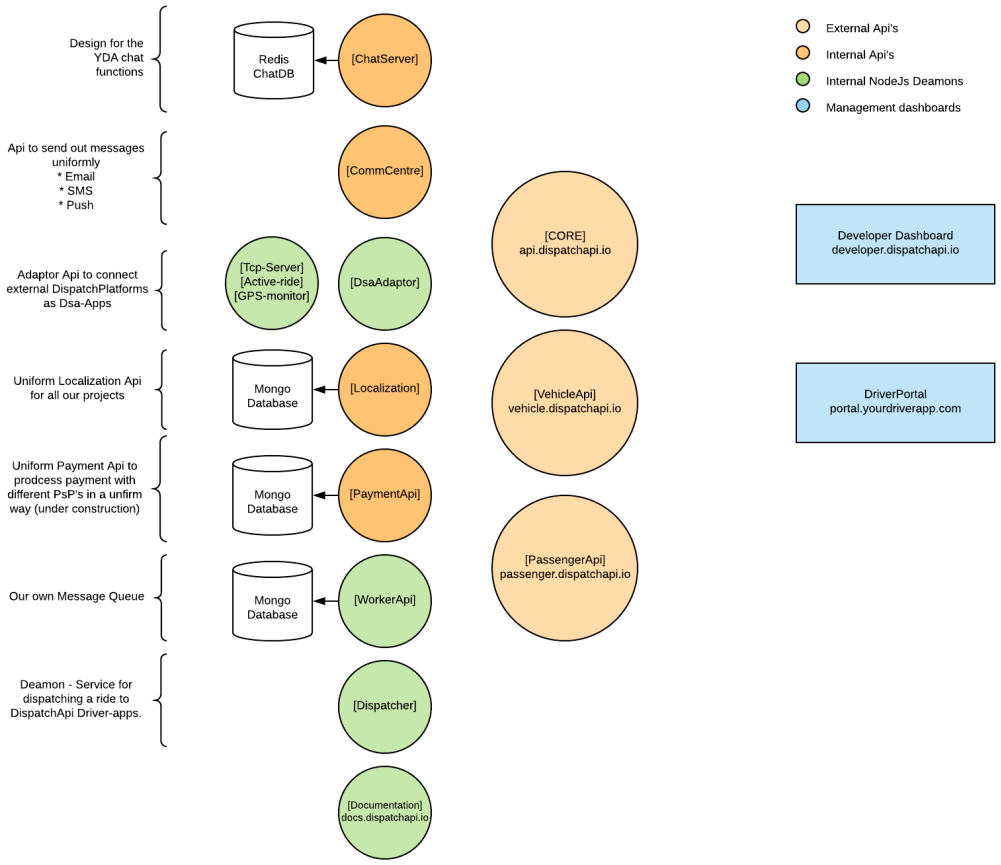


Fig. 3.1 Current System Architecture provided by taxiID.

The orange colored services are used internally, the green shapes are used by external partners. The smaller services adhere to the pattern that is called service-oriented architecture, where application components provide services over a network typically. Within the architecture, a separation exists between user interfaces, business logic and data storage, that is known as the three-tier or multi-tier architecture, as described in [16].

3.2.1 Monoliths

The bigger shapes in Figure 3.1 may be classified as monoliths. In the context of computer software, a monolithic system may have different definitions. Rod Stephens captures the meaning of a monolithic architecture quite broadly: "In a monolithic architecture, a single program does everything. It displays the user interface, accesses data, processes customer offers, prints invoices, launches missiles, and does whatever else the application needs to do" in [17]. In general, a monolith describes a software application which is designed without modularity. Even though the frontend is separated in some cases, it fits the description most

accurately. Integration of TPS could be achieved by implementing TPS as a component of a monolith. But what logically follows is either duplication, or dependencies between large systems. The first contradicts an important principle of software engineering; don't repeat yourself (DRY), the second limits scalability and independence of deployment. The legacy system has demonstrated this issue because it has its price calculation system implemented in this manner, now facing difficulties providing the price calculation functionality to newer projects.

3.2.2 Microservices

If the legacy price calculation system was implemented as a service, it could have been reused or replicated as a second separate price calculation system for YDA instead. A consensual definition of microservices does not exist, but can be defined as a development technique that structures a system architecture as multiple loosely coupled services, exactly opposing the description of a monolith. The smaller shapes in Figure 3.1 can be described as miniservices or microservices. Philipp Hauer describes the advantages of independent services accurately in [18], mentioning; improvements in development speed through parallel development, isolated deployment and continuous delivery (CD), scalability and potential parallelism, and independence in case of failure. Fair points of criticism have been made in regard to microservices. Jan Stenberg has pointed out that microservices are information barriers in [19], 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, security, and message formats.

3.2.3 Frontend and Backend

A model-view-controller pattern is separating the business and presentation layers in various frontend projects. This would mean that separate views have to be developed for each portal, or the views should be provided to the portals via iframes. In the last case, it may be beneficial to combine the frontend and backend in the same project structure. However, this would be in conflict with this three-tier pattern, which is not desired in respect to the evolution of the system architecture. Integration of the backend would mean that the core system should contain the price calculation system as a component, and separation of the backend would mean that the backend would be set up as a separate service. If this pattern is to be respected, four possibilities remain with which the frontend and backend could be implemented:

1. Integrate views in existing portal, build TPS as a separate microservice

2. Build separate service providing iframe views, build TPS as a separate microservice
3. Integrate views in existing portal, integrate TPS as monolith component
4. Build separate service providing iframe views, integrate TPS as monolith component

The final decision, based on a comparison seen in table 4.1.1 in Appendix A, proposes to separate the backend and integrate the frontend. This leaves the requirement of implementing the frontend in multiple portals simultaneously unresolved, but improves consistency of each individual portal implementation, and reduces dependencies of having to implement iframes.

3.3 Information Dependencies

The frontend separation or integration cases have little influence on the further design of the system. The backend separation case however, is only possible if information dependencies are satisfied. User identifying information must be retrieved from a system containing the source of truth. Company and product information must be retrieved from adjecent systems, or stored in the trip pricing system database. Important data that must be acquired are: products, companies, applications, users, settings and VAT amounts. In isolation, this model contains all the required information to calculate a price, if the parameters shown in Listing 3.1 are provided.

```
{
  "companyId": string
  "daAppInstallId": string,
  "vehicleTypes": string[],
  "passengerCount": number,
  "requestedDate": ISODate,
  "departure": { "gps": { "lat": string, "lng": string } },
  "destination": { "gps": { "lat": string, "lng": string } }
}
```

Listing 3.1 Minimal external information required for a trip price calculation.

The concrete data from the conceptual model could in theory be stored in one database, separate from the existing core database. How would the company data be synchronized? And does the system know which pricing rules should be used for the calculation? Assuming that companyId and daAppInstallId are provided in the authentication headers, the user can be identified. But this identity is futile if no pricing data is associated with it. There are three options with regard to storing the data in a way that user identity can be used to associate pricing information:

1. Centralized state - A single centralized database
2. Distributed state - Multiple distributed synchronized databases
3. Minimal state - Multiple independent databases with stateless references

A single source of truth, central database, would avoid having duplicate data all together. No synchronization, thus no network communication is necessary, and all data is always readily available. This design desecrates the independence aspect of microservices. The multiple synchronized databases in option 2, raises the problem of having duplicate, out of sync data. A one-way dataflow could reduce this problem, but it is not always entirely avoidable. This design does adhere to the concept of microservices by allowing independent deployments still. The minimal state option 3, would improve on the previous option 2, by only allowing data to be referenced back in a stateless manner. This means that a company in the core database could have related data stored in the TPS database, without being 'aware' of it. The entities that reference the company can be used in an autonomous fashion, where only the necessary information is sent to TPS whenever a request is made. Multiple proposal were made aiming to solve the combination of authentication, authorization and data consistency problems.

3.4 Authentication and Authorization

In the legacy system, authorization was achieved by sending extra headers for each crucial piece of information, this is clarified in Appendix A, chapter 3.4. To prevent data 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 data dependencies. In the slides of Appendix B.3 four examples options are listed as to which authentication could be implemented. These examples are based on three proposals listed in Appendix A, chapter 4.4, which are further explained in the subsections below.

3.4.1 OAuth 2.0

This authentication mechanism delegates user identity management 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. Although granting permissions should not necessarily be done by any user, this mechanism could be used for authentication purposes in the way that Facebook login would work as a central verification authority. The

token is verified by the authority authentication server on each request, keeping the user identity management centralized inside one single service.

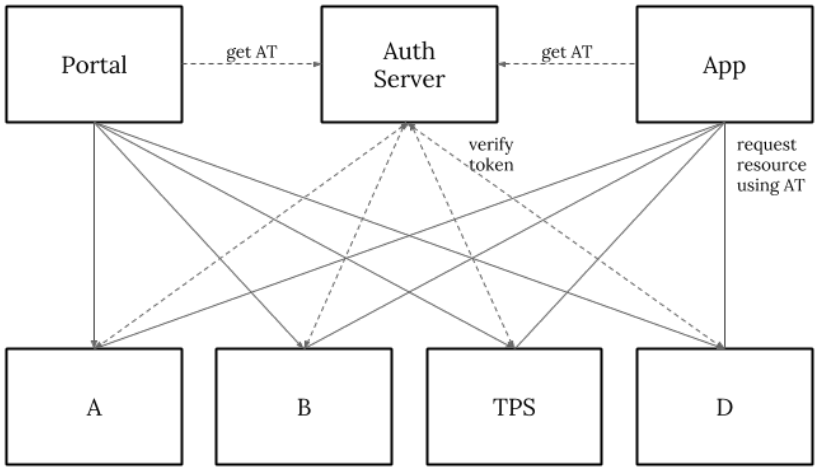


Fig. 3.2 OAuth requests where tokens are verified by Auth Server.

This proposal is used in example four of Appendix B.3.

3.4.2 JSON Web Token

This proposal 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.

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

Listing 3.2 Two user identifiers and registered claim names stored inside the payload of a JSON web token.

The keys other than companyId and daAppInstallId describe expiration date of the token, and other meta information. 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 full stop. The first part (header) of the token contains information about the hashing algorithm that is used to encrypt the payload. This part is Base64Url encoded. The payload itself contains information stored in JSON format as shown in Listing 3.2. The identity of the user is stored in the payload that can only be revealed by whoever holds the secret with which it is signed. Then the message can be verified using the third part of the token, which is the signature. The verification step prevents tampering with the payload. Claims can be added to the payload as shown in 3.2 to provide information about the token, as explained in [20]. Figure 3.3 adds statelessness to the previous proposal, thereby removing the verification step with the authentication server.

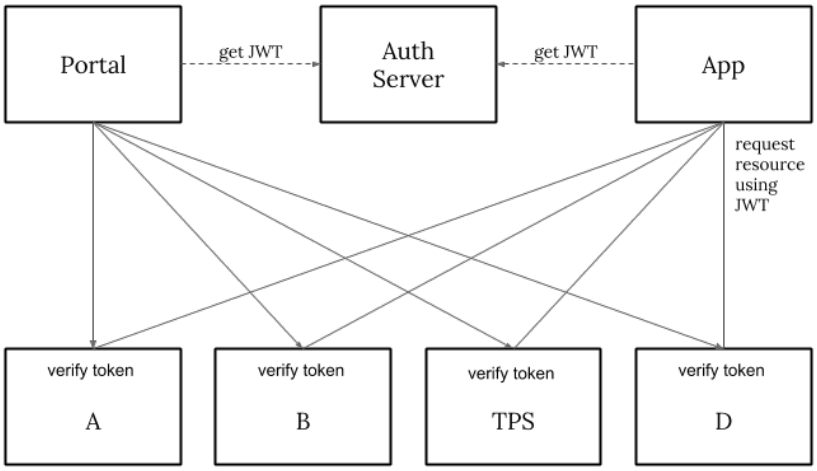


Fig. 3.3 OAuth with stateless JWT requests.

3.4.3 API Gateway

The final proposal allows services to be used by external agents via the API Gateway. This solution 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 [21]. 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. 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.

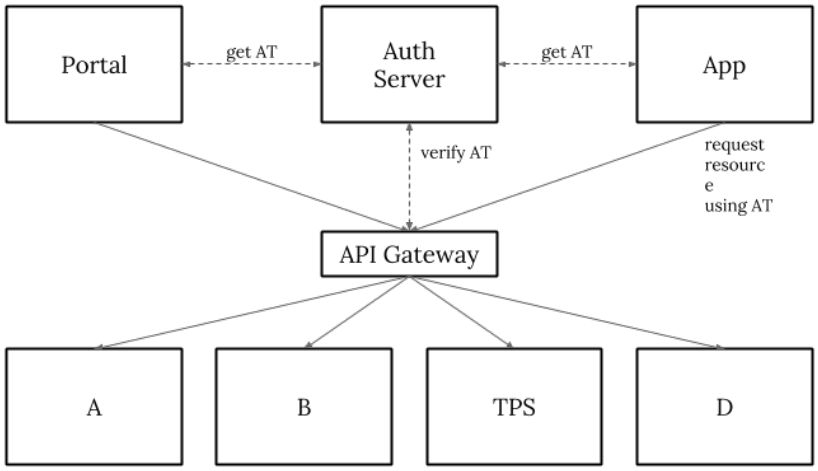


Fig. 3.4 API Gateway.

The fourth example is proposed as the best possible solution as shown in Appendix B.3. This solution would use the single responsibility authentication service as shown in Figure 3.2, and the minimal state option as shown in Figure 3.3. A successful login would yield a JWT from a dedicated authentication service. A database with a single source of truth would allow the authentication service to provide truthful user identifying information. The JWT would allow every system to acquire user identifying information from the token payload, and keep from having token verification round-trips. The only synchronization steps are executed when a new company or application is created, or when they are deleted. The proposal completely decouples the information requirements by keeping Core data and TPS data in their respective databases separated.

3.5 Methods and Techniques

All API projects in taxiID have been developed using JavaScript, with the exception of one legacy PHP project. Java and Swift are used for mobile applications. It is not beneficial to explore every single possible combination of technologies, as the range of possibilities is too big. But it is important to look at some popular alternatives. NodeJS offers more modern features and abilities to separate concerns in comparison with PHP. Speed and consistency of the codebase are important reasons to opt for a JavaScript NodeJS project, as advised in Appendix A, chapter 4.6. Two proofs of concept were made, one showcasing an Express

solution using GraphQL to expose resource, and the other exposing resources using Loopback in a hybrid JavaScript Typescript project.

3.5.1 Backend Framework

The first proof of concept allows consumers of the API to dictate the information that they want to receive. Using this concept, an API Gateway could easily chain requests, mapping resources from multiple services to a single endpoint. The proof of concept is available at github.com/Menziess/Typescript-GraphQL-API. This proof of concept was not advised to be used for TPS, because the inconsistencies between services introduce unnecessary complexity for developers. The first solution is vastly different from existing projects, which will introduce inconsistencies once again. For example, one API could make network requests to separate Loopback API's, but then has to implement a new inconsistent request format to deal with a GraphQL API. And this activity is repeated for all other depending API's and applications. For this reason, the second proof of concept has been chosen to start the project with. The team has experience with LoopBack 3.0 [22], enabling the team to reuse code, maintain their development velocity, and reason about this project more effectively. The project structure as shown in Appendix B.4 is made up of of Loopback configuration files, and Typescript files containing important logic. This separation is not ideal, but expresses the fact that some JavaScript files belong to a framework, and must adhere to a special framework format. The Typescript files offer more strict checks through static typing, interfaces and classes.

3.5.2 Frontend Framework

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. For the portal, a proof of concept was made for the case of having the portal implemented as a separate project. The proof of concept is available at github.com/Menziess/Typescript-Reiskosten. This concept was used to illustrate the differences between Vue and Angular, embodying mostly application structure. Angular being more suitable for large corporate application, while Vue caters toward smaller, more flexible, less structured applications. 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 YourDriverApp portal has been constructed using Angular 5. If the frontend is to be integrated, Angular will be the framework that is used to construct the views.

3.5.3 Database

Agarwal and Rajan state that NoSQL takes advantage of cheap memory and processing power, thereby handling the four V's of big data more effectively, but lacks robustness in comparison to SQL databases in [23]. Performance of geoWithin and geoIntersect queries have been tested between PostGIS and MongoDB. 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 lacks spatial functions that OpenGIS supports. Indexing of spatial fields is said to have a big impact on performance. The conclusion states that the downside of using NoSQL compared to SQL is the limitation in respect to spatial functions. The previous chapter discussed important features that were required for the location matching functionality to operate properly. Both OGC and GeoJSON standards offered sufficient support. In the paper of Schmid et al. 2015 [24], the team argues that clustering is much easier in MongoDB, which may be important in the future when a company grows. In respect to the CAP theorem, and ACID properties, SQL and MySQL have different strengths and weaknesses:

	SQL	NoSQL
Integrity	✓✓	x
Scaling	x	✓✓
Atomicity	✓	✓
Consistency	✓✓	✓
Isolation	✓	✓
Durability	✓	✓
Availability	✓	✓✓
Partition Tolerance	✓	✓
Performance	✓	✓✓
Maturity	✓✓	✓
JSON Documents	x	✓
Clustering	✓	✓
Sharding	✓	✓✓

Table 3.1 Comparison between SQL and NoSQL databases.

Performance and scalability are important properties for taxiID's systems. MongoDB has the ability to scale horizontally. Because MongoDB has good sharding capabilities, location

related performance issues may be solved by setting up local database systems. NoSQL document storage allows for great horizontal scaling and sharding, catering more towards use cases that do not require overall consistency of the data, but makes the data highly available. Performance of spatial queries is very important, for which the conclusion of Agarwal and Rajan support the use of NoSQL.

3.6 Conclusion on System Architecture

What is most fitting solution to integrate the backend and frontend into the existing architecture?

A NodeJS loopback microservice should be implemented along with a MongoDB database, resulting in a scalable high performance solution that can be deployed independently. Introducing a stateless authentication service to implement identity management, allowing the microservice to be more decoupled, bringing the amount of verification requests down to zero. This solution is most fitting to the existing architecture, that only has to implement one future proof authentication method.

Chapter 4

Trip Price Calculation System

4.1 Introduction

The term 'rule-based' in the title caters to the proposition that trip price calculations hinge on information defined as something called a rule. This chapter contains resolutions that are less based on empirical evidence and more on the input of the product owner and a balancing of arguments that support important quality attributes. The questions in regard to the implementation of the backend are answered, resulting in a system that deterministically calculates trip prices using rules that are restricted by user defined criteria.

4.2 The System Structure

In the previous chapter, the second proof of concept was mentioned that separates the calculation logic from the Loopback framework, as shown in Figure 4.1. The isolation of the calculation and direction classes result in a more robust system. The Price object is composed of a Directions and a Calculator Class, both of which expose only one method, directions and calculate respectively. The behavior of the gathering of directions information and calculation of prices is encapsulated in different subclasses using the strategy pattern as described in [25]. Allowing behavior to change on demand. The entities stored in the database are conceptualized in Figure 4.2, and will be referenced throughout this chapter. Chapter three concludes that MongoDB is the proper database for this system. MongoDB allows relations to be embedded in the parent's document, which is useful for child entities, like timeframes, that are never shared.

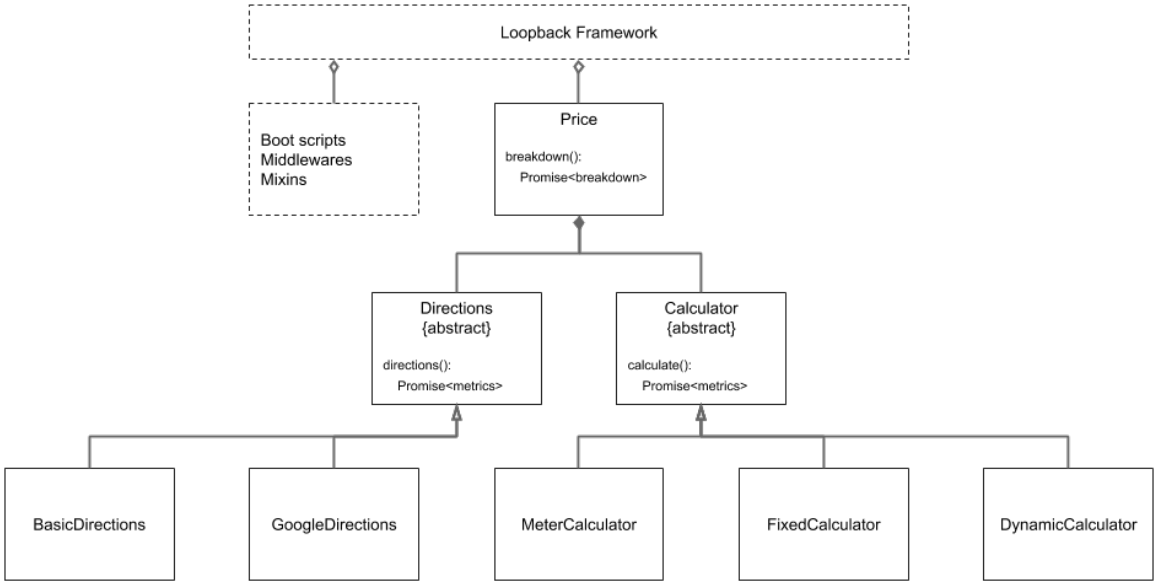


Fig. 4.1 High level class diagram.

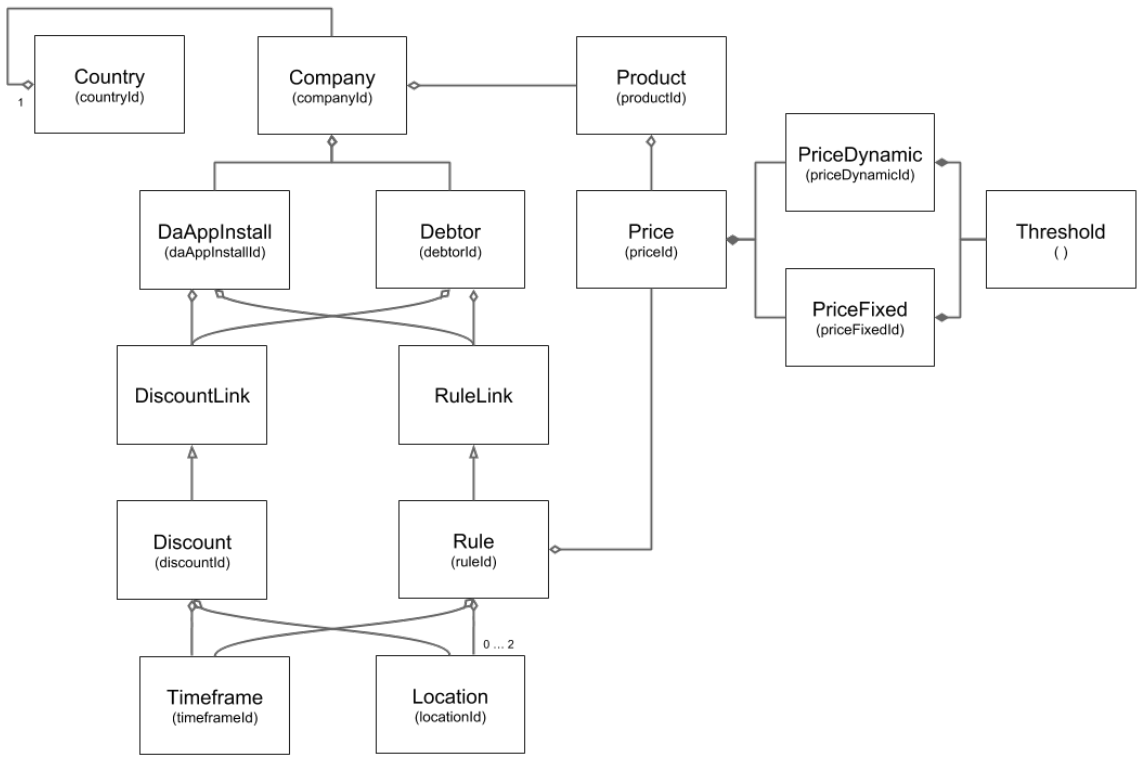


Fig. 4.2 Conceptual data model showing database entity relations.

4.3 Matching Criteria

A rule is the rudimentary element of the trip pricing system that allows users to interpret and define the way with which prices are assigned to trips, restricted by the dimensions of space and time. In the legacy system, discounts were a fundamental part of the price definitions. Meaning that whenever a price was found, the discount was directly associated with the price. This design led to some issues, mainly revolving around having to duplicate prices in order to have the same prices without discounts. It is mutually beneficial for users and developers who are destined to maintain this system, to create a system that offers a lot of freedom in applying criteria to prices. In perspective, a user may want to apply the following restrictions:

1. Define certain prices for one app, but not for the other
2. Define lower dynamic prices in Rotterdam
3. Define a certain fixed price during New Year's Eve
4. Assign a discount for trips that depart from Schiphol
5. Define a higher price for trips that end in larger cities
6. Make the limousine available only in North Holland
7. Allow for free trips between two companies during the weekend

There are also some restrictions that are applied by default as pre- or postconditions. For example, the passenger count should not exceed that of the products' passenger capacity. An on demand option in the booking app allows passengers to book a ride without providing a destination. In these cases, a rule must still be matched with the departure location, and the user may want to disable some products for the on-demand functionality. This is an example where the products are shown without prices, or filtered out if the user wishes it. Compared to the legacy system, this design of allowing criteria to be applied to rules and discounts independently, resulting in a more modular system with more freedom. The user is allowed to define discounts and rules separately, and when a new concept is added that makes use of locations or timeframes, the user is able to reason about it in the same fashion. For the maintainers of this system, it is beneficial to have less duplication, more separation of concern, and less overhead.

4.3.1 Locations

Chapter two concludes that MongoDB's geometry datatypes are sufficient location encodings. The location matching examples can directly be implemented as a basis for the location matching criterion. The listed restrictions that a user may apply, depend on two locations

being matched or ignored simultaneously. On top of that, one best match must be selected from the matching results, which must be done by determining what the solution for location overlap should be.

The Conventional Approach

The legacy postal code and address based system maintains a type based matching order: fixed, tier, and dynamic rule types. A location conflict would not occur if the user was able to make distinct pricing records. If it were to occur, the system would simply pick the first match. But with the introduction of overlap, this will no longer be feasible. North Holland contains Amsterdam, which contains Amsterdam-Centrum, which contains the Dam Square, which may contain a pickup location defined by the user. There exists some difference in magnitude of locations in the new system, whereas the old system had locations of the same magnitude.

The New Approach

Multipolygons allow for multiple locations to be defined as one, opening up the possibility of defining all branches of a company in a single location, which could then be selected as departure and destination location. This would solve example seven from the examples list, using just one rule and one location. Because locations could be associated with all rule types, all location based examples are easy to define. Although care must be taken to distinguish between not having a location defined, and not providing a location when booking a ride. If no departure location is assigned to a rule, the rule should match with any location. And if the passenger orders an on-demand ride, where the destination is not provided, the destination should be ignored during the matching process.

```
const query = [];

if (departure) {
  query.push({
    $match: {
      $or: [{
        "departure.area": {
          $geoIntersects: {
            $geometry: {
              type: "Point",
              coordinates: [
                departure.gps.lat,
                departure.gps.lng
              ]
            }
          }
        }
      },
      {
        "departure": { $exists: false }
      }]
    }
  })
}

db.collection('Location')
  .aggregate(query)
```

Listing 4.1 Matching departure.

In Listing 4.1, the query is built conditionally. If a location is not provided by the booking app, it will not be evaluated. Only the provided locations are matched if they intersect with a defined destination, or if the destination does not exists in the database. This covers the cases for checking only one of the two locations. In chapter two, solutions have been proposed to the overlapping locations problem. From all the interesting approaches, the most straight forward solution is simply assigning a priority number to the rule. This solution has the advantage of interpretability and flexibility. The behavior of the matching system can easily be tinkered with by the user. The user can reason about the fact whether one rule has a precedence over the other by comparing the priorities.

4.3.2 Timeframes

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. This either means that the timeframe one window of time, or that each given day has a single window of time.

But if a discount should be active during night of New Years Eve, between 23h and 5h, this description would not be sufficient to cover this use case under any interpretation.

The Conventional Approach

The legacy system takes a straight forward approach of storing time in a relational database. The begin and end of a window are stored in a record that is related to a parent timeframe entity. The timeframe has many windows that could contain a timestamp. It either finds one or many time windows that contain the timeframe. This approach covers all possibilities imaginable. The downside of this approach is the complexity to interpret or mutate the value of the timeframe.

The New Approach

For this reason, a proposal was made to implement timef rames in a way that let users choose to describe each hour of the week, being stored as a bit map. The windows could be decreased to half an hour, resulting in twice as many bits. Three implementations have been tested, where the bit string format offered the best outcome, as seen in B.6. A timeframe is stored having two ISODates (international standard: ISO 8601), and a bit string representing the schedule for which the insert statement is shown in Listing 4.2.

```
db.Timeframe.insert({
  startDate: new Date(2018, 4, 7),
  endDate: new Date(2019, 4, 7),
  weekSchedule:
    "001101000110011011000011
    011010110011000010111100
    101010101110100011111000
    111110011111011100100001
    101000000010111011100100
    110010000001000010101101
    010111101000000101001110"
})
```

Listing 4.2 Improved timeframe.

A string is a very flexible datatype. Using a regex in a query makes checking multiple bits in the string relatively easy, and enables different values next to 0 and 1. 3. A bit array would only allow for 0 and 1 to be used. A bit string also makes querying the data really stable, as the query will simply not match if the content of the data is not of expected length or value. Performance is not an issue if the regex column is indexed, and when prefix expressions

(/^/) are used, as per documentation in [26]. As noted before, the system is easy to scale if existing data can be migrated to deal with a new amount of bits, or new character usage over bits.

```
/**
 * Date object days start at sunday, in order let monday be
 * index 0, decrease the index by one, but limit numbers
 * in the range of [0, 7).
 */
const startMonday = (d: number) => (d - 1) % 7;

/**
 * Creates a regex that spreads bits across hours of each
 * day of the week.
 */
export const regexFromDate = (date: Date) => {

  const skip =
    // Day of the week multiplied by hours a day
    startMonday(date.getDay()) * 24
    // Hour of the day
    + date.getUTCHours();

  return { skip, timeRegex: new RegExp(`^.{${skip}}1`) };
};
```

Listing 4.3 Opening timeframe.

The regexFromDate could be used to create a regex that could be used in a query to check whether a single hour within a week is set. Skip is an integer representing the number of bits that should be skipped to get to the moment represented by the date. So in order to get 11 AM - 12 AM in the presented schedule, 3 * 24 skips + 11 skip = 83 skips are to be made to find the digit 1 on thursday. Because the getDay method on JavaScript date objects return an integer resembling the day, starting at sunday, the startMonday function is used to pretend that it starts on monday.

4.4 The Trip Price Calculation

The process from start to end is has many edge- and corner cases. Three major stages of the process: handling the incoming request, finding the matching price rule, and calculating the prices, are explained concisely in the following subsections to provide a general overview. Important details of calculation types will be expanded upon in later sections of this chapter.

4.4.1 Incoming Request

The flowchart in Figure 4.3 shows the point where the request is received, up until the point where enough information is known to fetch rules and discounts from the database. When the request is received (1), the user is authenticated. The JSON Web Token contains the user identity, the companyId and daAppInstallId (2). The request body contains information about the ride: vehicle types, passenger count, requested date, departure, and destination.

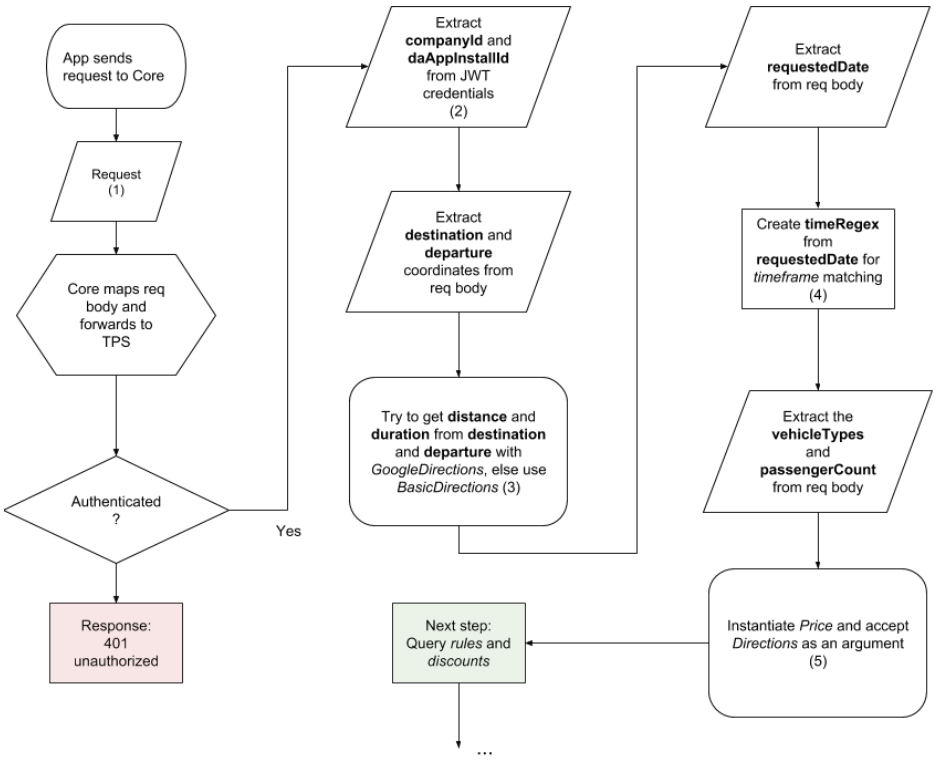


Fig. 4.3 The condensed flow of a trip price calculation - incoming request.

The Directions class will provide an interface to retrieve trip related data. The departure and destination are fed to the Directions class, which will proceed and work out the distance and duration of the trip (3). If the GoogleDirections class is unable to determine the trip details, the BasicDirections class returns a base case result. The trip price calculation flow changes drastically when no destination or departure locations are provided in the request body. Having alternative behaviors helps dealing with providing the most accurate information possible. The strategy pattern also improves the systems resistance to change. If a different service is needed to determine the trip details in the future, it can easily take the place of one of the current services. If the trip details have not been obtained, but at least one of the two, departure or destination locations, have been provided, a database query could still work out the best matching price rule based on partial information. The requestedDate is

to be converted to a regex pattern (4) for reasons explained in the timeframes section of this chapter.

4.4.2 Data Aggregation

When the user is authenticated, the system immediately requests the distance and duration of a ride by providing the departure and destination locations to the directions service. This service awaits the trip details response while it is fed to a Price class instantiation (5). The Price class will wait until matched pricing rules are provided, upon which it will perform the price calculation. Before this is the case, the aggregate queries are performed (6), trying to find a matching rule and discount for a particular company/app combination. Two separate queries start by finding the application and company combination for which a price is calculated. If a reference to a debtor is provided, rules and discounts linked to it are used instead. The rules and discounts contain criteria in the form of timeframes and locations. A discount has basic properties while the rule has complex pricing information for products. Figure 4.4 shows the most important stages of the rule aggregation pipeline. The discount aggregate is a more simple version and has some of the same stages, its flowchart is therefore excluded.

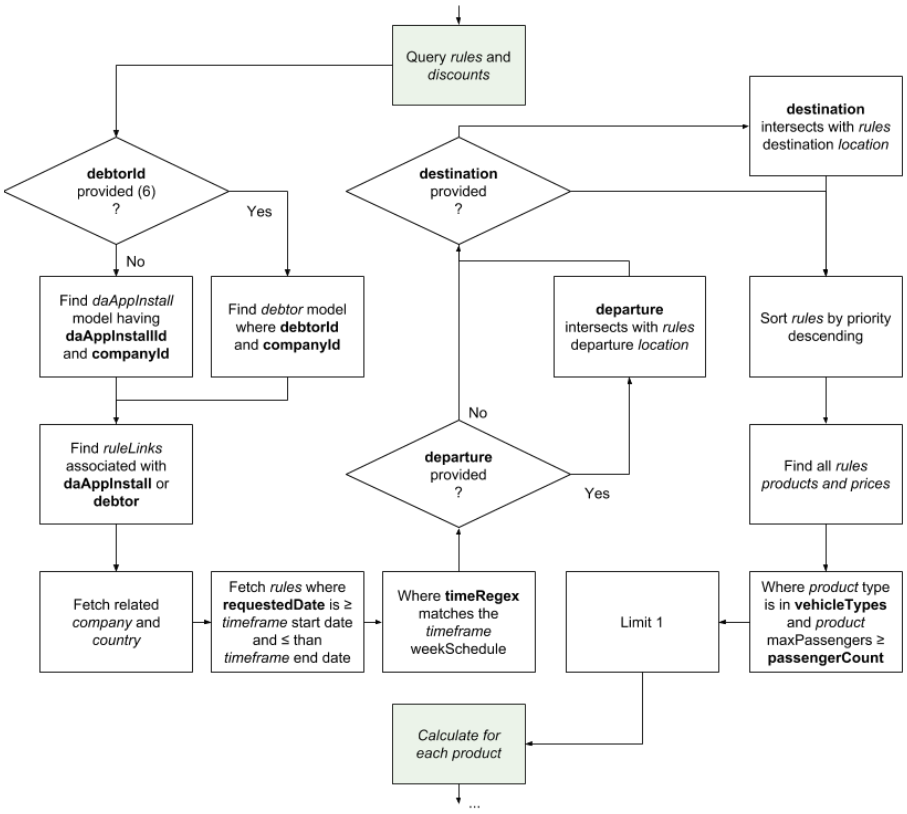


Fig. 4.4 The condensed flow of a trip price calculation - data aggregation.

4.4.3 Calculation

When both queries have finished, potential discounts are added to each pricing rule, which are then fed to the Price class asynchronously. A single rule has price information for each available product of that rule. If a company offers three products, it is possible to only offer two products in a given timeframe or area by associating them with a rule. For each product that is related to the price rule fetched from the database, a price breakdown is calculated. If the array of matched rules is empty, a map over the array will result in an empty array of price breakdowns. Pricing information is validated before the calculation is started using the method shown in Listing 4.4. The system should throw an error, as a price calculation can not proceed without the required information.

```
/**
 * Check if pricing contains valid properties and is not undefined.
 */
public static validPricingOrError(pricing: pricing | undefined): void {
  if (pricing === undefined) {
    throw new HttpError('Pricing data is undefined.');
  }
  const missing = [
    'prices',
    'rules',
    'country',
    'company',
    'type',
    'maxPassengers',
  ].filter(prop => !(prop in pricing));
  if (missing.length) {
    throw new HttpError('Pricing data is missing properties:\n\t' + missing);
  }
}
```

Listing 4.4 Find missing properties.

A price breakdown follows a series of logical steps, in which functions are created that have a certain default behavior. The strategy pattern is once again applied to use the appropriate calculator for the type of calculation that is required. The different types of calculations are discussed in the next chapter.

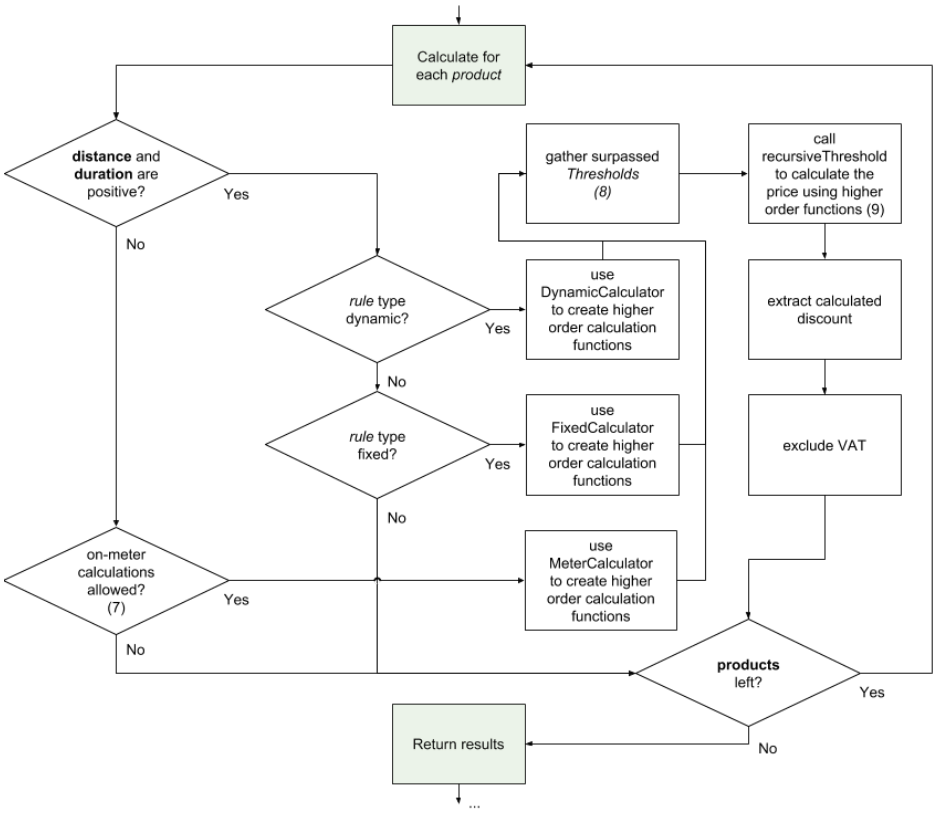


Fig. 4.5 The condensed flow of a trip price calculation - calculation.

In step (7) the system faces the issue of not being able to calculate a price, because the distance and duration are not known. In this case, the user is able to choose whether on-meter calculations are allowed to be made, meaning that a meter is used to capture these metrics at a different period of time. If the user agrees, the products are returned with a truthy isAllowedOnMeter flag. The system will still follow the calculation procedure, but will use the MeterCalculator instead of the other two. Step (8) and (9) will be expanded upon in the Threshold Calculations chapter.

4.5 Price Calculation Types

The calculate method of each calculator subclass receives the pricing information of a product and directions information of a ride. This setup proves to encapsulate all possible calculations that depend on any metric type (distance, duration, ...) and any more complex operation. The calculator method will perform the calculation in a fixed number of steps:

1. Define the metrics (and units) for the calculation

2. Define a lambda function expressed a blueprint to generate a collection of calculator functions
3. Generate calculator functions that can be used by a deterministic function
4. Call the deterministic function
5. Return the total price

```
/**
 * Generate functions based on prices and units, incorporating
 * default prices that are passed next to different metrics
 * and stored in a indexed object which names are types.
 */
public static generateCalculators(
  prices: indexed,
  metrics: metrics & indexed,
  func: Function,
): object {
  const calculators: indexed = {};
  for (const type in metrics) {
    calculators[type] = (
      price: number = prices[type],
      unit: number = metrics[type],
    ) => func(price, unit);
  }
  return calculators;
}
```

Listing 4.5 Calculator generator.

The generateCalculators method generates a function for each metric with default parameters.

4.5.1 Dynamic

A dynamic calculation is performed on the metrics: distance and duration, having units: kilometer and minute. A straight forward lambda expresses that the units are multiplied by the price. From this expression, calculators are generated. When a calculator is called without arguments, the default parameter values are used. When the calculator function is called with just the distance, the second default parameter will be the price for that distance. This property is needed for the threshold calculations that are discussed in the next section. The thresholds array contains pricing information for each successive threshold that has been surpassed.

4.5.2 Fixed

Opposed to a dynamic price, the fixed price can not be expressed as a quadratic formula. However, thresholds may increase the price in steps.

4.5.3 Meter

The meter calculator should return zero under every circumstance. The on-meter calculation simply returns a breakdown for which all values are zero. This is done by convention at taxiID, so that the mobile apps can still display the products that are available, and determine the price at a later stage.

4.6 Threshold Calculations

On top of each calculation type, prices can be defined after certain thresholds are surpassed. For example: if a taxi travels 25km, a threshold could be defined at 20km, after which the price will be 10 cents cheaper. In this case, the passenger pays a normal price for the 20 kilometers, and a cheaper price for the last 5 kilometers. This procedure is executed for each metric, but only if the cascading option is specified. Only the name of the metric used to measure thresholds and the actual calculation functions may change. For this reason, it is possible to define a function that recursively walks through all surpassed thresholds, then calculates a price using different calculations after each threshold was surpassed, as shown in Listing 4.6

```
/**
 * Recursive function that calculates a price for each threshold
 * in a way specified by a calculation function that is passed
 * until the thresholds are empty.
 */
public static recursiveThreshold(
  thresholds: threshold[],
  calculation: Function,
  metric: number,
  cascaded: boolean = true,
): number {

   if (thresholds === undefined || thresholds.length < 1) {
     log('\tno more thresholds at ' + metric + ', using default price');
     return calculation(undefined, metric);
   }

   log('\tthreshold met at ' + metric);

   if (!cascaded) {
     return calculation(thresholds[0].value, metric);
   }

   const nextMetric = thresholds[0].threshold;
   const newMetric = metric - nextMetric;
   const price = calculation(thresholds[0].value, newMetric);

   return price + Price.recursiveThreshold(
     thresholds.slice(1),
     calculation,
     nextMetric,
   );
}
```

Listing 4.6 Recursive threshold calculation.

The Typescript type definitions reveal that calculation is of type Function, and thresholds is of type threshold[]. The base case returns the calculation with an undefined first argument. This forces the calculation method to use its default value, which is actually the normal kilometer or minute price. The base case will have the value of the first threshold that is met, assuring that the passenger pays the normal price up until that point. If the base case is not satisfied, the function checks whether the cascading boolean is true. This boolean determines whether each threshold should be evaluated, or only the last one. In case of the fixed price calculation, only the last threshold fixed price will be computed. But for the dynamic prices, each step has to be added to the total amount. Finally, the calculation is made using the next threshold, and the recursive call is summed up with the calculated price.

4.7 Determinism

A deterministic algorithm always produces the same output given a certain input. This concept is paramount when calculating prices. This does not imply that a criterion is not allowed to change. It does mean that given a certain input and a certain set of criteria, the output should be the same every single time. External state or time are crucial factors that determine whether a function can be deterministic. The concept of determinism is further explained in a later chapter. In the object-oriented programming (OOP) paradigm, state is managed through encapsulation. Object instances manage their internal state, and expose public methods to allow certain changes to be made on those instances. The functional programming (FP) paradigm treats computations as evaluations of mathematical functions. This aspect fits perfectly with the goal of reaching determinism. Using FP in conjunction with OOP, a higher order structure may be implemented using classes and encapsulation for more flexibility, while minimizing or eliminating state mutations through pure functions. Parts of the system may execute operations asynchronously, emphasizing the importance of managing state flawlessly. There are factors that can not be controlled also, such as computations of external services at different times. For example, if a passenger books a ride, and Google's algorithm gives a different estimated distance and duration, the price may go up or down. These factors are side effects that are out of the scope of containment. Side effects that can be contained should be eliminated as much as possible. Professor Frisby demonstrates side effects in [27, Chapter 3]. The functions slice and splice both being pure and impure respectively, have equal results in the first operation on an array. But the second time the method is called, splice returns different results because of side effects. The recursiveThreshold static method is called for every price related calculation. It is honest about its parameters, and does not depend on external state. It is pure, because it does not depend on state outside of its scope, and does not mutate the data that is passed to it. The function is deterministic because when given a valid input, it produces the same output every single time. The method merely executes the passed function with different arguments. The base case will therefore always yield the default result. The elegant nature of recursion counts as an inductive proof that the procedure will work for each successive step, which works in harmony with the uncertain amount of thresholds that need to be processed. Because the method is pure, it can easily be tested. Unit tests have been written using the Mocha framework [28] and the Chai assertion library [29], to cover the most important aspects of the system, making sure that the price calculation logic keeps producing the same outputs. Everything that has been noted so far, should apply to all the functionalities in the calculation system to keep the process deterministic. To further reduce the chances of introducing bugs, some additional techniques could be used, including Software Validation

techniques. Linting is the process of running a program that will analyse code for potential errors. Static analysis tools may be used to find code smells. Continuous Integration may be implemented to ensure that valid builds are deployed. The current setup of calculation and directions subclasses allow for extensibility with respect to the open-closed principle. Metrics, calculators, threshold types, price definitions, and other aspects of the calculation may be added as long as the same interfaces are used. State and mutations should be fully encapsulated using OOP patterns, leaving only static functions exposed. Functions should be written in a functional style so that no state is changed outside of the function scope, and that the function is absolutely honest about its parameters and return values. Typescript plays an important role in mixing OOP and FP together through type definitions. This deterministic nature of TPS will mean that a list of breakdowns as result is always guaranteed.

4.8 Breakdown

The final result of the trip price calculation is a breakdown for every requested product. For example: if the mobile application requests prices for 'saloon' and 'limo' vehicle types, the response will at most contain an array with two breakdowns, for saloon and limo products. To ensure a seamless transition from the legacy price calculation system to TPS, the response formats should be identical. Still an improvement, if profitable enough, could be taken into consideration. One requirement of the price breakdown states that the tax should be included, but as shown in Listing 4.7 the included tax is part of the breakdown. Is it by mistake or design?

```
[
  {
    "price": {
      "currency": "EUR",
      "total": 850,
      "breakdown": {
        "route": 802,
        "tax": 48,
        "toll": 0,
        "parking": 0,
        "waiting": 0,
        "discount": 0
      }
    }
  },
  ...
]
```

Listing 4.7 Legacy price breakdown

Two possible solutions were proposed having VAT included in the price. The first solution extracts the tax element from the breakdown, so that the sum of the breakdown would add up to the total price where VAT is included in the price as shown in Listing 4.8. As demonstrated in Appendix B.2, a breakdown is easily constructed in four steps when VAT is included.

```
[
  {
    "price": {
      "breakdown": {
        "route": 8300,
        "toll": 0,
        "parking": 0,
        "waiting": 0,
        "discount": -1650
      },
      "currency": "EUR",
      "total": 6650,
      "tax": {
        "amount": 400,
        "percentage": 6
      }
    }
  },
  ...
]
```

Listing 4.8 Improved price breakdown

Keep in mind that unlike the listings the prices in the proposal are not displayed in cents. The second solution maintains the legacy format, but has to recalculate the prices without VAT. This could have downsides unlike the first approach:

1. If an error is detected in the calculation, it is hard to trace back which components contributed to the total VAT. This would be even harder when each component uses its own VAT percentage.
2. It takes extra steps to calculate the price of each component excluding VAT.
3. Rounding the individual components could result in a sum that is not equal to the total displayed in the breakdown.

The first proposal has been implemented, resulting in an operational trip price calculation system.

4.9 Conclusion on Trip Price Calculation System

Which logic and data is required in the backend to reliably calculate a trip price?

Locations should only be used as criteria if the relevant location coordinates are provided by the booking app. Timeframes, passenger count, vehicle types, on-meter and estimation options always apply as criteria by which a rule is matched. The original rules are implemented to each have their own classes, producing deterministic higher order functions that calculate the final trip price, using functional programming techniques.

Chapter 5

The Portal

5.1 Introduction

The term ’reasoning’ in the title, meaning "the action of thinking about something in a logical, sensible way", may have been redundant if a system were to calculate trip prices autonomously. But this system is designed with the user in mind. That is why the inner workings of the system must be expressed in such a way that allows the user to predict the outcome after changes have been made to pricing rules with confidence.

5.2 Visual Hierarchy

A pricing rule is a set of criteria that covers pricing information. These criteria and pricing information are stored in database entities. These entities could be expressed as view components. For example, "the pickup location of a trip must be located in Amsterdam" is a criterion that is stored in a location entity, that may be expressed as polygon on a world map component.

5.2.1 Essential Components

The main entities that make up a the price calculation system are visualized in Figure 5.1. The plurality of the child entities describe whether more than one child are present within the parent entity. For example, rules have many products, with each product having one price, which has one priceFixed and one priceDynamic.

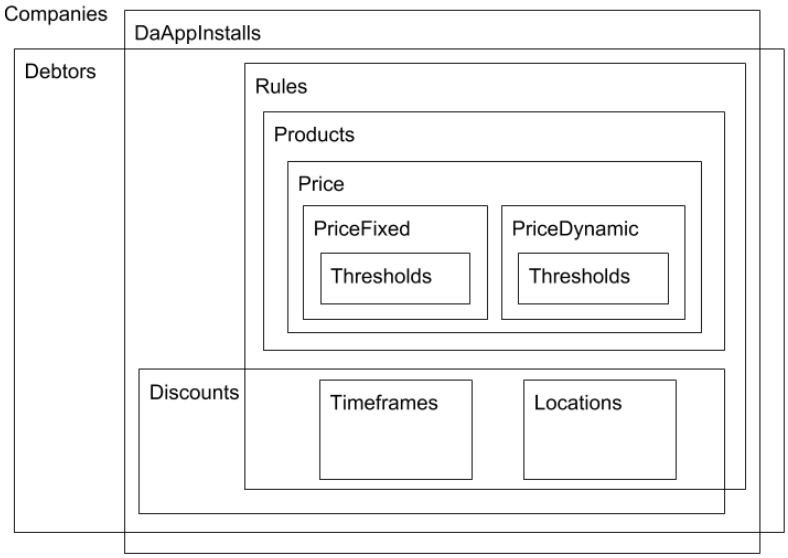


Fig. 5.1 Treemap of entities.

5.2.2 Expressing Order

A human brain is capable of ordering the differently sized boxes by enclosure, proximity, intensity, and spatial grouping. Stephen Few states that "Perception of these basic visual attributes is called 'preattentive' processing, in contrast to the conscious part of perception, which is called 'attentive' processing. Preattentive processing is extremely fast and broadband in that we can simultaneously perceive a large number of these basic visual attributes, called 'preattentive attributes'. Preattentive perception is done in parallel, but attentive processing is done serially and is, therefore, much slower." in [30, p. 3]. This fact can be utilized to construct a hierarchy similar to that of Figure 5.1 through visual queues. A famous phrase often used in data visualizations called Shneiderman's mantra [31], lays the foundation of principles that enable users to maintain an understanding of the context in which data is visualized. The sentences in the mantra dictate that there are three stages in data exploration. This mantra in combination with preattentive attributes could be put to good use in reducing the cognitive overhead while reasoning about pricing rules.

1. Overview First
2. Zoom and Filter
3. Details on Demand

5.3 Design

Composing a pricing rule should require no level of cognitive effort. It should be linear process where few choices result in the desired outcome. This could be achieved by separating related reusable entity components from the pricing rule. For example, a location does not have to be defined in the pricing rule view. It could be defined in its own separate view, so that it can later be selected, allowing the user to only reason about a subset of criteria at once. This theory works for reusable entities such as locations, but would result in a negative outcome when used with information that belongs to a single pricing rule, such as dynamic or fixed prices. The contextual understanding is lost when the user has navigate to a different view, while keeping locations, time, rule type, and other properties in memory while filling in the different price properties for different products. The mantra could be applied to offer an overview of a particular entity. The user could then select an entity and modify the details. Displaying the properties on a single page, ordered using preattentive attributes. The portal follows a basic hierarchy:

1. Overview: enumerates over a list of entities
2. Detail page: contains one particular entity
3. Composite page: displays a combination of lists of entities and/or single entities that belong together

Entities that are part of a many-to-many or one-to-many relation should have their own overview and detail pages, otherwise the user may think that a particular entity that is combined into another entity's view, may exist for that entity only. If the user modifies that entity, it has consequences for all other relationships with that entity. For example, if a location was shared between many rules, the user could edit the address of the location, unaware of the consequences that the change brings to all the rules that depend on that particular location. One-to-one or many-to-one relationships do not have that problem. For example, thresholds are embedded, meaning that they are not part of any entity other than the one that embeds them. With this concept, the following entities should have their own overview and detail pages: Products, Locations, Rules, Discounts, and DaAppInstalls. The mantra could also be applied to the representation of entity properties in the same fashion. But for each enumeration, a direction of space on the page must be filled. As a webpage only has two dimensions of space, the properties that fill up the space must be grouped in some fashion. Preattentive attributes can be utilized to illustrate that some elements on composite pages belong to each other, or are in fact distinct.

5.4 Products

The sidebar contains links to the overview pages. The products overview page shows a searchable and sortable list of products. These products may be selected, upon which the user is brought to the detail page where the type, name and other properties can be set, and where the product may be deleted. When properties have changed, the save button becomes available. And when the user tries to leave the page after changing a property, a prompt is shown that allows the user to leave or continue editing a product. These features have been implemented for all components.

5.5 Pricing

The pricing overview contains a table with two tabs: 'rules' and 'special rates'. The word 'discounts' may imply that it would only be able to lower the prices of trips. Special rates can be negative or positive, and can be expressed in percentages or fixed amounts. Discounts and rules have a lot in common. They both have locations and timeframes, they can both be prioritized, and they both have an impact on price calculations.

5.5.1 Discounts

On the discount detail page, the name, priority, type and value properties may be set. The discount type can be changed between 'fixed' and 'percentage', changing the symbol next to the value from a € to a % symbol. The timeframes and locations can be found below the normal properties. For the timeframes a component has been created that can be integrated into any form. Timeframes consist of two dates between which a discount or rule is active. A date picker allows users specify the date, and a separate time input allows users to specify the time. When users want to define more complex timeframes, a specific week days option opens up an hour selection element. When this option is enabled, the time inputs are hidden. Only one option can be enabled so that no confusion between the time aspect of the timeframes can exist. When this toggle is set, the time aspect of the start and end date is hidden, as from that moment, time can only be defined as individual hours. Because this aspect of the timeframe is stored as a bit string, it is easily translated into an html input element, and can easily be modified in the future. The location selector component can be found right above the timeframe selector component. It has a 'from' and a 'to' input in which the user may enter a location by name or address. Autocompletion results allow the user to quickly find a one of the company defined locations. When the user presses enter and the location does not exist, the user is prompted to create a new location. All locations that have been

created by developers may be shared, so that the user is able to clone them. This is added as a requirement in a later stage, allowing users to create pricing rules without having to bother with locations first. When the input is empty, the 'Everywhere' option is selected, which ensures that no location is associated with that input.

5.5.2 Rules

On top of the functionalities just covered, the rule detail page has more complex property inputs. A table of columns for each product is displayed with a minimum price, waiting price, start price, kilometer price, and minute price inputs on each row. The kilometer and minute price can be extended through thresholds. When a threshold is added, it copies the values from the row above, which may then be edited. The timeframes and location selectors look the same as those of the discounts, but only when the rule is of type 'dynamic'. When the user selects the 'fixed' type, the form is transformed. The columns in the table only contain a waiting price and fixed price input. The location input is moved next to the fixed price input. The locations can no longer be set to 'Everywhere' to avoid expensive mistakes. A button allows prices to be extended with subrules, just like the way kilometer and minute prices could be extended with thresholds. A visual hierarchy exists between the different components on the page through proximity and spacial grouping. The views can be seen in Appendix B.9.

5.5.3 Priority

The priority for discounts and rules can be defined in the priority input field, although modifying this field would be inconvenient for large amounts of modifications. A drag and drop feature was added to sort more quickly.

5.6 Locations

Locations have been defined as either points or areas. In technical terms, this is sound. For an average user, this does not sound intuitive. That is why locations are named as two groups: locations and areas. The technical solution in matching points was initially based on points being distinct coordinate pairs, which could only be precisely matched if the user selected a location from the location service that was never changed, like Google Places. The locations found in searches would exactly match the coordinate pairs, as that is where the points are originated from. But when a passenger drops a pin on the map, it would have very little to no chance of matching the exact coordinate pair. That is why all locations are stored as

polygons initially, with a coordinate pair denoting the centroid of the location. The polygons may be imported, drawn and edited by the user in a later stage.

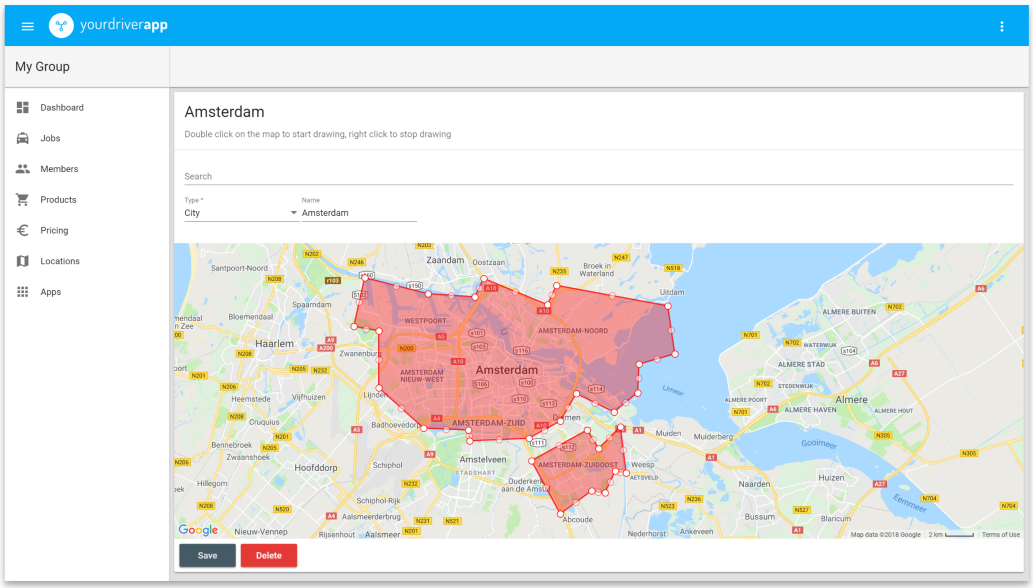


Fig. 5.2 Amsterdam - A single area comprised of multiple locations.

5.6.1 Point

When a point is created, only a search bar is visible initially. Autocomplete suggestions from Google Places are shown when the user starts typing. These results are filtered to only include addresses and smaller locations, to maintain the distinction between areas and points. A shape class has been created that is capable of plotting points, polygons and multipolygons on a Google Map using a method called autoDetect. When the user selects a location, the gps coordinates are used as a center around which a polygon is plotted.

```
/**
 * Generate a polygon around lat lng coordinates.
 */
private static fromPoint(
  lat: number,
  lng: number,
  theta: number,
  radius: number,
) {
  const points = [];
  const p = (lat, lng, x, y, r) =>
    [lat + (r * x), lng + (r * y) * 1.5];

  let x, y;
  let angle = 0.0;

  while (angle < 2 * Math.PI) {
    x = length * Math.cos(angle);
    y = length * Math.sin(angle);
    points.push(p(lat, lng, x, y, radius));
    angle += Math.abs(theta);
  }

  // Closing the polygon.
  points.push(points[0]);
  return Shape.fromPolygon(points);
}
```

Listing 5.1 Generating Polygon From Point.

When a point is created, a user searches for a place in Google Places, gives it a name and a descriptor, and saves it as if it were a string of characters that would be matched within the system. After a point has been created, it may be modified by dragging the indices of the polygon. New polygons can be added by double clicking on the map, and edit mode can be stopped by right clicking on the map.

5.6.2 Area

The area detail page contains a map right at the creation screen. A search bar allows users to find a location, and add it to the map. The user may draw and modify polygons before the location is saved. The biggest difference between locations and areas is the search service that powers the autocomplete search. When a location is selected in the area search, a polygon is imported from OpenStreetMap as seen in Appendix B.8. Although this feature is great because it gives users the choice to avoid drawing polygons all together. The browser will have a hard time rendering all the indices of such complex shapes.

5.7 Apps

After the user has created locations, products, special rates and rules, nothing will happen. When a new rule or discount is saved, it is not activated by default. The app detail page allows rules and discounts to be associated with an application, upon which they become active. For pricing rules, extra settings may be set that allow applications to show price estimations, or calculate the trip price using a taxi meter. From that point, the user is able to order a ride using the companies' booking app to test whether the expected outcome is the consequence of the changes.

5.8 Conclusion on The Portal

Is it possible to communicate the inner workings of the system through the user interface?

The sidebar reflects which major concepts exist within the portal: Products, Pricing, Locations, and Apps. Composite views reflect the coherence that exists between related entities, while separation of independent entities reduce the cognitive overhead while reasoning about pricing rules. Spatial grouping on the page hint which properties belong to which entity, for which all human beings have a natural intuition. Manual activation and sorting of rules force the user to actively decide whether a rule should match, which can be tested immediately using the companies' booking app.

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 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 price calculation 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. See Appendix B.1 for the sprint review. Before the beginning of the sprint, different techniques and project structures have been tried, providing a head start.

6.3 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. In Appendix B.3 a proposal has been made to solve authentication for TPS. Authentication has

been implemented in this sprint using JSON Web Tokens that are signed by the core system. This implementation saves a lot of maintenance for TPS and other projects that adopt this token format in the future. The token stores two important identifiers that are structurally used in taxiId projects: companyId and daAppInstallId. 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. Another proposal was made to format the breakdown correctly, as shown in Appendix B.2. 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. These basic steps have been implemented during this sprint, as shown in the sprint review Appendix B.4. Finally, the project is deployed to a staging environment so that the system could be used by the applications in the staging environment for testing.

6.4 Sprint 3 - Products and Pricing

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, adding to the preliminary tasks of 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 is only operational in the backend, so the task of displaying thresholds in the frontend is moved to the next sprint. Appendix B.5 shows some screenshots of the earlier implementations. Lastly, the calculation classes have been refactored to be more robust.

6.5 Sprint 4 - Apps and Timeframes

Feedback is given by the product owner after each sprint, resulting in new requirements and modifications to requirements. A functionality is desired that enables users to sort pricing rules and special rates, by dragging the rows in a table to the correct positions. Whenever a priority changes, all subsequent rows need to be updated in order to maintain a consistent

prioritized list without duplicate priorities. Another requirement will enable products to be returned in the breakdown as 'on-meter' results. This means that, whenever a destination or departure location is undefined, the system will return products without a price, so that the apps can assign a price later, but will still be aware of the available products. A view is added that displays all apps of a company, and a detail page is added in which rules and discounts can be associated with those apps. This detail page is created in three short iterations. All titles, labels and other texts are replaced by references to a localization api for internationalization purposes. Timeframe components are added to multiple views, in which hours per week can be specified between two dates. For this feature, a proposal was made that is shown in Appendix B.6.

6.6 Sprint 5 - Thresholds

Thresholds, just like prioritized rules and discounts, should be ordered, as they are independently embedded in multiple price entities that are to be synchronized consistently. A rule has pricing information for every product, and every product has many thresholds. Whenever a threshold property is mutated, every other threshold must be updated as well. Also no duplicates are allowed, and no empty price values are allowed inside the thresholds. Thresholds can independently be removed and added, where in the last case values will be copied from the last to the newly added threshold so that the user will not need to manually insert all values. The timeframes are expanded so that either hours per week are used to determine whether a rule should trigger, or a time should be set to further restrict the begin and end date of the timeframes. Price estimations are added as a setting on all relations between all entities and rules, or discounts. Authentication is implemented in the portal so that a JWT is requested at the core system. The token is stored in localstorage and used to communicate with TPS. The automatic call to generate synchronized company data is enabled for the core system, such that every time a company is created in the core system, it is also created in TPS.

6.7 Sprint 6 - Locations

An overview for locations has been added, showing points and areas. A point could be created by searching for a place in Google Places. An area is a larger, more complex area, that should be imported as a more rich polygon. An implementation using OpenStreetMap was able to import complex locations and plot them on a Google Map. Appendix B.8 shows the distinction between the two import functions. After locations have been imported, the

user is able to edit the shapes and properties of the location. Because the OpenStreetMap polygons contain far too many indices, the user is confronted with a shape that would take far too long to modify. For this reason, the OpenStreetMap feature was removed from the project. In exchange, the Google Places service point is used as a basis to draw a polygon around. The size of the polygon that is drawn depends on the viewport that Google Places provides, so that larger areas will import larger polygons. Rule subrules had to be added in the rule detail page, for the user to be able to edit multiple rules at once. This issue was not resolved during this sprint, and was moved to the next sprint.

6.8 Sprint 7 - Subrules

Sub rules were initially estimated to take up two days of development. Each time a sub rule was added, it had to be activated for the same apps as the parent rule. Whenever the parent rule changed its type from 'fixed' to 'dynamic', all sub rules had to be deactivated somehow. Whenever a parent rule ws deleted, all relations had to be deleted in a cascading fashion. And whenever a parent rule's priority changed, the ordering that had to be maintained had to keep the child rules separately updated. This task of micromanaging small properties, keeping data consistent, caused this task to take twice the estimated story points. Appendix B.9 shows the first implementation of subrules. An authorization mixin was added so resources no longer have to be restricted to certain owners manually.

6.9 Sprint 8 - Polishing

The final major feature changed the way users were able to access predefined locations. An extra tab shows relevant locations defined by taxiID that can be copied and modified in a preview screen. This allows users to set up their locations more quickly. The remaining time is dedicated to fixing bugs and processing feedback.

Chapter 7

Conclusion

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

The four challenges manifested in the research questions that constructed this assignment have been answered in order to successfully implement the trip price calculation system. The best possible way of mapping conveyable locations to pricing rules in a system that integrates in the existing system architecture of taxiID, keeping the original rule types while improving the interpretability of the matching logic through the user interface, has been researched. Information available in the public domain has been used to answer empirical questions, from which proposals based on good arguments have been made, and working proof of concepts have been created, out of which some have been implemented, resulting in a system design that could be used for many other purposes in the software industry.

The best way of mapping

Which location encoding is sufficient for this system to be operational? How can legacy location definitions be improved to be universally interpretable? In what way can location matching be improved? Which Database Management Systems are candidate for handling this project's use cases?

Addresses and postal codes can be translated to geometric datatypes such as Points, Polygons and MultiPolygons. Geometry based locations can be visualized, and thus interpreted regardless of the country in which a location resides. Matching is done through the OpenGIS or GeoJSON API, by writing geospatial queries depending on the selected candidate database system. Selected candidate database systems are systems that adhere to the OpenGIS or GeoJSON standard, yielding many possibilities, of which MySQL and MongoDB have been proven to be workable. The problem of overlapping locations, can be solved using complex

and straight forward approaches, thus proving this location encoding to be sufficient for this system to be operational.

What is most fitting solution to integrate the backend and frontend into the existing architecture? Which architectural patterns fit in with the existing system architecture? How is state shared and synchronized between system components? What is the most applicable authentication method?

A NodeJS loopback microservice should be implemented along with a MongoDB database, resulting in a scalable high performance solution that can be deployed independently. Introducing a stateless authentication service to implement identity management, allowing the microservice to be more decoupled, bringing the amount of verification requests down to zero. This solution is most fitting to the existing architecture, that only has to implement one future proof authentication method.

Which logic and data is required in the backend to reliably calculate a trip price? Which criteria should regulate whether rules match? How can determinism of price computations be guaranteed? In what way can the three original pricing rule types be implemented? (fixed, dynamic, and threshold prices)

Locations should only be used as criteria if the relevant location coordinates are provided by the booking app. Timeframes, passenger count, vehicle types, on-meter and estimation options always apply as criteria by which a rule is matched. The original rules are implemented to each have their own classes, producing deterministic higher order functions that calculate the final trip price, using functional programming techniques.

Is it possible to communicate the inner workings of the system through the user interface? Which backend concepts are essential to display in the frontend? Which design practices allow users to understand coherence of different elements that make up a rule? How should a user know what the outcome of his interactions with the system are?

The sidebar reflects which major concepts exist within the portal: Products, Pricing, Locations, and Apps. Composite views reflect the coherence that exists between related entities, while separation of independent entities reduce the cognitive overhead while reasoning about pricing rules. Spatial grouping on the page hint which properties belong to which entity, for which all human beings have a natural intuition. Manual activation and sorting of rules force the user to actively decide whether a rule should match, which can be tested immediately using the companies' booking app.

References

[1] U. T. Inc. (2011) The uber story. [Online]. Available: https://www.uber.com/en-NL/about/

[2] (2018) Mysql 5.7 reference manual - spatial data types. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/spatial-types.html

[3] C. Sheldrick, "Efficient and future-proof." April 2018. [Online]. Available: https://what3words.com/2018/04/read-our-white-paper-efficient-and-future-proof/

[4] (2016) what3words - restful api. [Online]. Available: https://docs.what3words.com/api/v2/#autosuggest-params

[5] M. Schneider, "Spatial data types." [Online]. Available: https://www.cise.ufl.edu/~mschneid/Research/papers/Sch09BoChb.pdf

[6] ——. Conceptual foundation for the design and implementation of spatial database systems and gis. [Online]. Available: https://www.cise.ufl.edu/~mschneid/Service/Tutorials/TutorialSDT.pdf

[7] 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

[8] (2004) Geographic information – simple feature access – part 1: Common architecture. [Online]. Available: https://www.iso.org/standard/40114.html

[9] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, and T. Schaub. (2016) The geojson format. [Online]. Available: https://tools.ietf.org/html/rfc7946

[10] L. Xiang, , X. Shao, and D. Wang. (2016) Providing r-tree support for mongodb . [Online]. Available: https://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XLI-B4/545/2016/isprs-archives-XLI-B4-545-2016.pdf

[11] K. Holmberg, "On using openstreetmap and gps for optimization," Linköping Institute of Technology, SE-581 83 Linköping, Sweden, Tech. Rep., November 2015. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:867759/FULLTEXT01.pdf

[12] O. Tech. (2017) Nominatim usage policy. [Online]. Available: https://operations.osmfoundation.org/policies/nominatim/

[13] (2018) Postgis 2.4.5dev manual. [Online]. Available: https://postgis.net/docs/manual-2.4/using_postgis_dbmanagement.html#PostGIS_GeographyVSGeometry

[14] (2018) Mysql 5.7 reference manual - geometry class. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/gis-class-geometry.html

[15] (2018) Geospatial query operators — mongodb manual 3.6. [Online]. Available: https://docs.mongodb.com/manual/reference/operator/query-geospatial/

[16] 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

[17] R. Stephens, *Beginning Software Engineering*. John Wiley & Sons, 2015.

[18] P. Hauer. (2015) Microservices in a nutshell. pros and cons. [Online]. Available: https://blog.philipphauer.de/microservices-nutshell-pros-cons/

[19] J. Stenberg. (2014) Experiences from failing with microservices. [Online]. Available: https://www.infoq.com/news/2014/08/failing-microservices

[20] N. S. M. Jones, J. Bradley, "Json web token (jwt)." May 2015. [Online]. Available: https://tools.ietf.org/pdf/rfc7519.pdf

[21] M. Palladino. (2019) Microservices & api gateways. [Online]. Available: https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway

[22] (2018) Loopback 3.x - loopback documentation. [Online]. Available: https://loopback.io/doc/en/lb3

[23] K. R. Sarthak Agarwal, "Analyzing the performance of nosql vs. sql databases for spatial and aggregate queries." September 2017. [Online]. Available: https://scholarworks.umass.edu/cgi/viewcontent.cgi?article=1028&context=foss4g

[24] W. R. Stephan Schmid, Eszter Galicz, "Performance investigation of selected sql and nosql databases." June 2015. [Online]. Available: https://agile-online.org/conference_paper/cds/agile_2015/shortpapers/68/68_Paper_in_PDF.pdf

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[26] (2015) Mongodb evaluation query operators. [Online]. Available: docs.mongodb.com/manual/reference/operator/query/regex/#index-use

[27] P. Frisby, *Professor Frisby's Mostly Adequate Guide to Functional Programming.*, 2018. [Online]. Available: https://mostly-adequate.gitbooks.io/mostly-adequate-guide

[28] (2018) Mocha - the fun, simple, flexible javascript test framework. [Online]. Available: https://mochajs.org

[29] (2018) Chai assertion library. [Online]. Available: http://chaijs.com

[30] S. Few, "Tapping the power of visual perception." September 2004. [Online]. Available: https://perceptualedge.com/articles/ie/visual_perception.pdf

[31] "A thousand-fold increase in human capabilities." 1997. [Online]. Available: http://www.ifp.illinois.edu/nabhcs/abstracts/shneiderman.html

List of figures

2.1 Latitude Longitude on a Sphere 7
2.2 Amsterdam Drawn Polygon 10
2.3 Square Containing One Point 12

3.1 Current System Architecture 20
3.2 Proposal OAuth 2.0 24
3.3 Proposal Stateless JWT 25
3.4 Proposal API Gateway 26

4.1 Class Diagram 32
4.2 Data Model 32
4.3 Incoming Request 38
4.4 Data Aggregation 40
4.5 Calculation 42

5.1 Treemap of Components 52
5.2 Amsterdam Drawn Polygon 56

List of tables

1.1 Legacy CSV Pricing Definition File 2
1.2 Project Roadmap 5

2.1 Location Matching Requirements 11

3.1 Databases Comparison 28

Appendix A

Pregame

Online document: https://goo.gl/ZcJ8Aj

Pregame

Phase I - Pregame

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

subject Afstudeerstage bij TaxiID

year 2018

Author: Stefan Schenk
500600679

TaxiID
05-02-2018 te Medemblik

Tutor: Willem Brouwer

Hogeschool van Amsterdam
Software Engineering

Index

1. Introduction **3**

2. Requirements Specification **4**

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

3. Definition **12**

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

4. Solution **18**

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

5. Conclusion **32**
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

6. References **34**

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

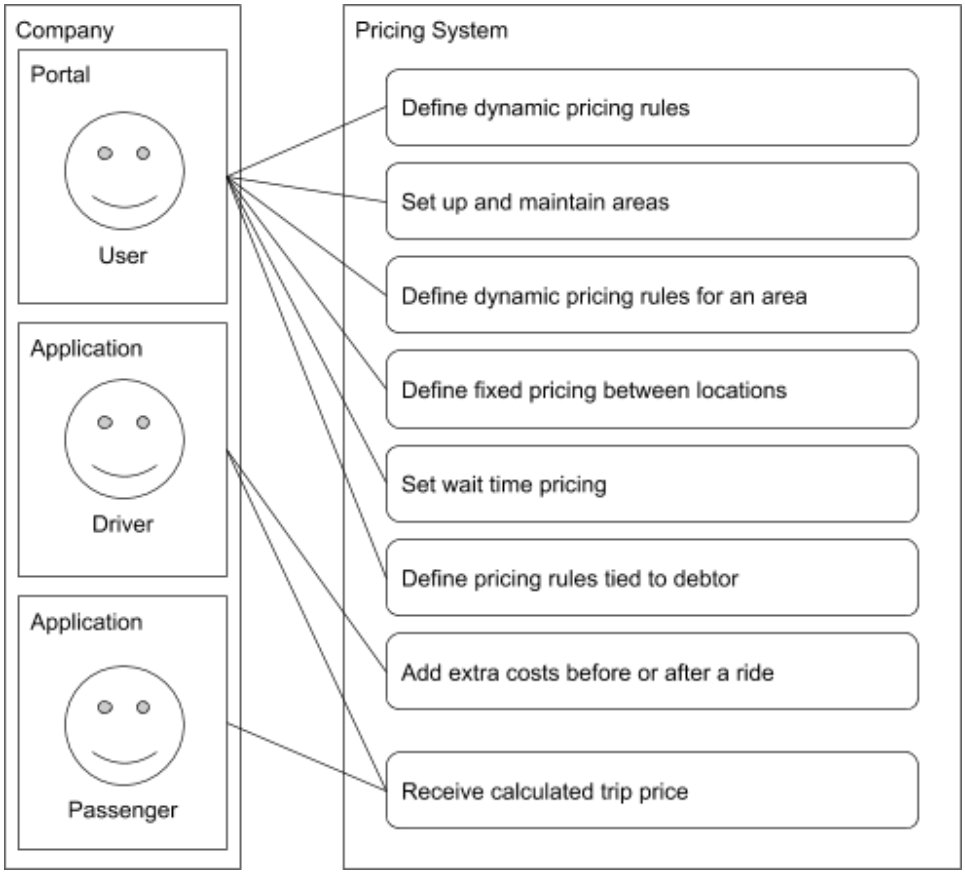


Image 2.4.1 - Use case diagram.

2.5. Requirements

2.5.1. Non-functional Requirements

ID	Non-functional Requirement
NFR1	For a logged in driver portal user (yourdriverapp.com or white labeled build) the solution should be seamlessly integrated in the portal.
NFR2	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.
NFR3	The prices should be attached to a DaAppInstall.

2.5.2. Functional Requirements

ID	Functional Requirement
FR1	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. This price should be calculated taking into account: 1. Starting rate 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 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 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.
FR2	A user can define a price per minute for waiting time, the spent wait time can be sent by the driver.
FR3	As a user I want to select areas from a predefined list to set up fixed price calculations.
FR4	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.
FR5	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.
FR6	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.
FR7	A driver can add positive or negative additions to the cost of the ride at any point in time. - Percentage (discount)

	- Driver defined (toll, parking, other) - Variable (waiting time - it has to be calculated inside the system, from an input of time)
FR8	It should be possible to set up a price with time constraints only (hire a limo) - this is just a dynamic rule
FR9	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

Bulk:	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).
CD:	Continuous Delivery / Deployment.
CI:	Continuous Integration.
Company:	A company that owns Applications.
DaAppInstall	An application installation.
Debtor:	A person or company responsible for the payment of a ride, on upon which the pricing can depend.
Driver Portal:	Portal that brings information from diverse sources.
Discounts:	A discount that is either a percentage, fixed amount or reference to rule containing prices.
Location:	A zip code or geometric location.
ORM:	Object Relational Mapping.
User:	A person, group or company that owns applications.
Passenger:	Uses an Application to order a taxi ride.
Product / Application:	An application bought by the User to which data is tied.
Pricing Rule:	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.
taxiID Partner Portal	Portal that brings information from taxiID sources.
Timeframe:	A collection of start and end times + days of the week.
Zones / Regions:	Polygons drawn on a map.
Core API:	Available through Developer Dashboard (developer.dispatch.io).
Passenger API:	Available through Passenger App.
Vehicle API:	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	1. Debtor identifier is sent to the API 2. The API checks if a debtor identifier is sent, and it exists in the database 3. The API tries to match the pricing rules that are tied to the debtor by: a. Departure location b. Destination location c. Ride time 4. A rule is found, the API tries to find a discount that is tied to the debtor based on: a. Departure location b. Destination location c. Ride time

	5. A discount rule is found 6. The fixed price is calculated with the discount

ID	2
Description	Passenger app doesn't send debtor identifier, a pricing rule is found, discount is found
Basic Flow	1. Debtor identifier is not sent to the API 2. The API checks if a debtor identifier is sent, it isn't 3. The API tries to match general pricing rules tied to the company by: a. Departure location b. Destination location c. Current time 4. A pricing rule is found, the API checks whether a discount is available that matches: a. Departure location b. Destination location c. Ride time 5. A discount is found 6. The fixed price is calculated with the discount

ID	3
Description	Passenger app doesn't send debtor identifier, a pricing rule is found, multiple discount are found
Basic Flow	1. The API tries to match general pricing rules tied to the company by: a. Departure location b. Destination location c. Current time 2. A pricing rule is found, the API checks whether a discount is available that matches: a. Departure location b. Destination location c. Ride time 3. Multiple discount are found 4. The discount rule with the highest precedence is taken 5. The fixed price is calculated with the discount

ID	4
Description	Passenger app doesn't send debtor identifier, a pricing rule is found, no discount are found
Basic Flow	1. The API tries to match general pricing rules tied to the company by: a. Departure location b. Destination location c. Current time 2. A pricing rule is found, the API checks whether a discount is available that matches:

	a. Departure location b. Destination location c. Ride time 3. No discount is found 4. The fixed price is calculated

ID	5
Description	Passenger app doesn't send debtor identifier, a pricing rule isn't found, discount is found
Basic Flow	1. The API tries to match general pricing rules tied to the company by: a. Departure location b. Destination location c. Current time 2. A pricing rule isn't found, the API tries to find a dynamic price rule 3. A dynamic price rule is found, the API checks whether a discount is available that matches: a. Departure location b. Destination location c. Ride time 4. A discount rule is found 5. The fixed price is calculated with the discount

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	1. The API tries to match general pricing rules tied to the company by: a. Departure location i. The point is found in the area in the database b. Destination location i. The point is found in the area in the database c. Ride time i. The timeframe contains this ride time 2. Multiple rules are found that match locations and timeframe a. The rule with the highest precedence (highest number) is picked to calculate the price 3. A discount is not found, the price is calculated

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	1. The API tries to match general pricing rules tied to the company:

	a. Departure location i. The gps location is found in polygon collections of rule A and B b. Destination location i. The gps destination location is found in rule A and B c. Ride time i. The timeframe contains this ride time 2. Multiple rules are found 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) 3. A discount is not found, the price is calculated

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 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.

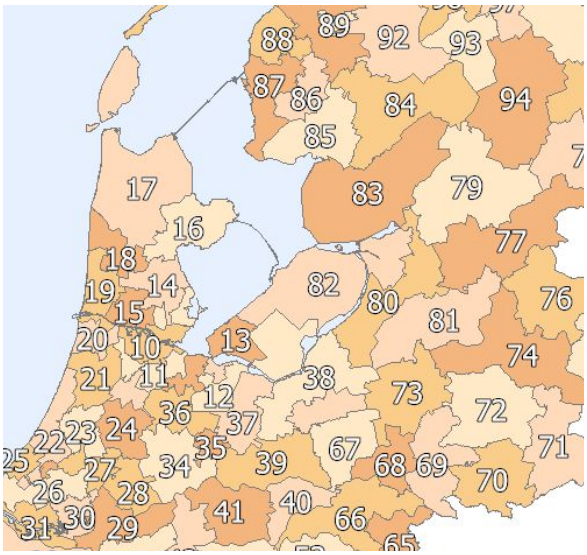
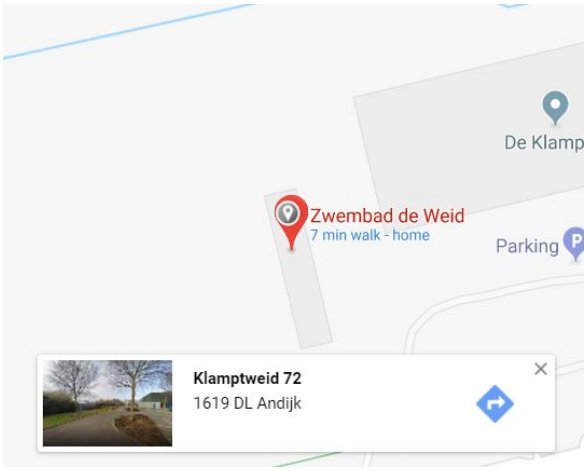
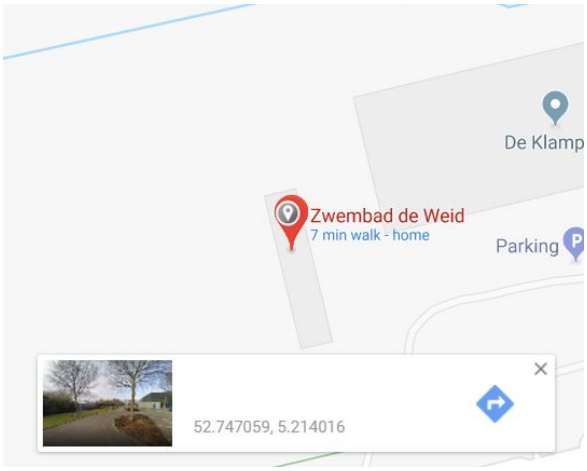
3.2. Functional Requirements

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
	
Collection of zip codes to define an area	Polygon on a map to define an area
	
A single point defined by street,street nr, zip code city name	A set of gps coordinates with a range to define a single point

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 DaAppInstalls. 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.

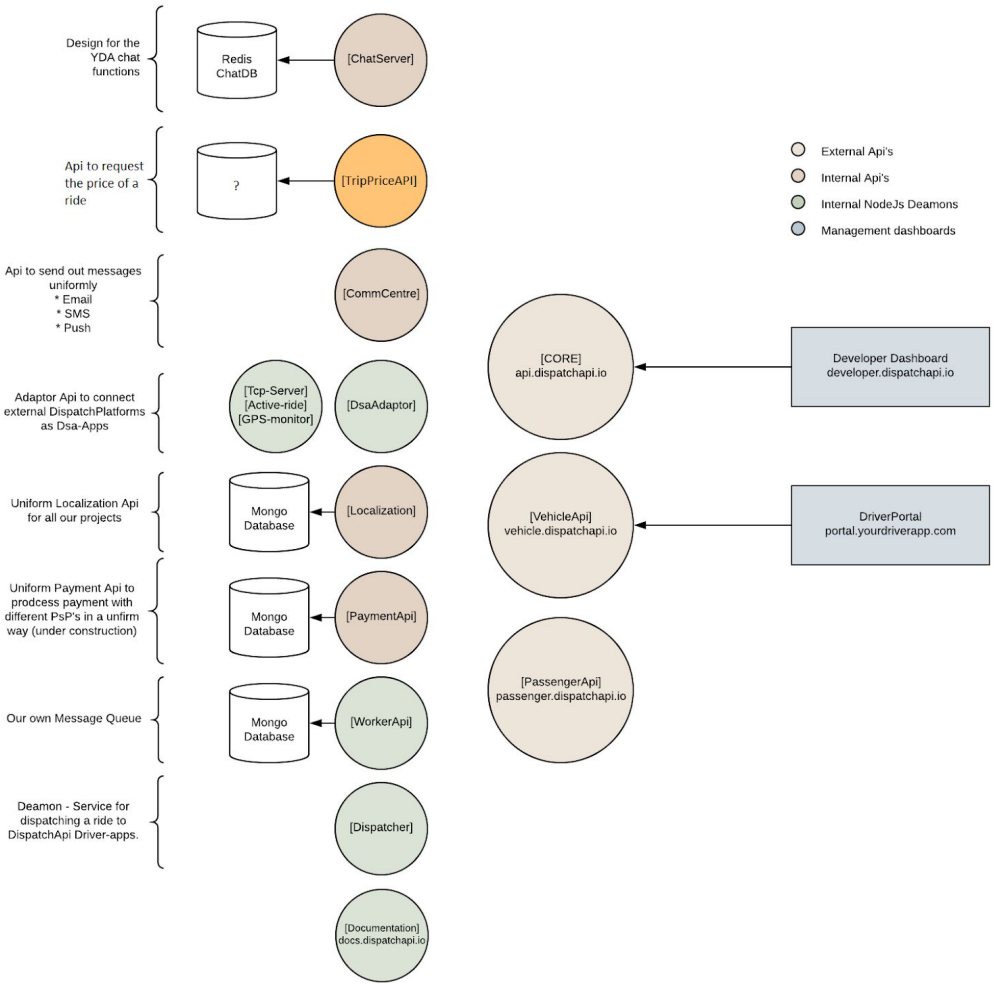


Image 3.3.1 - Current system architecture of the dispatch api.

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.

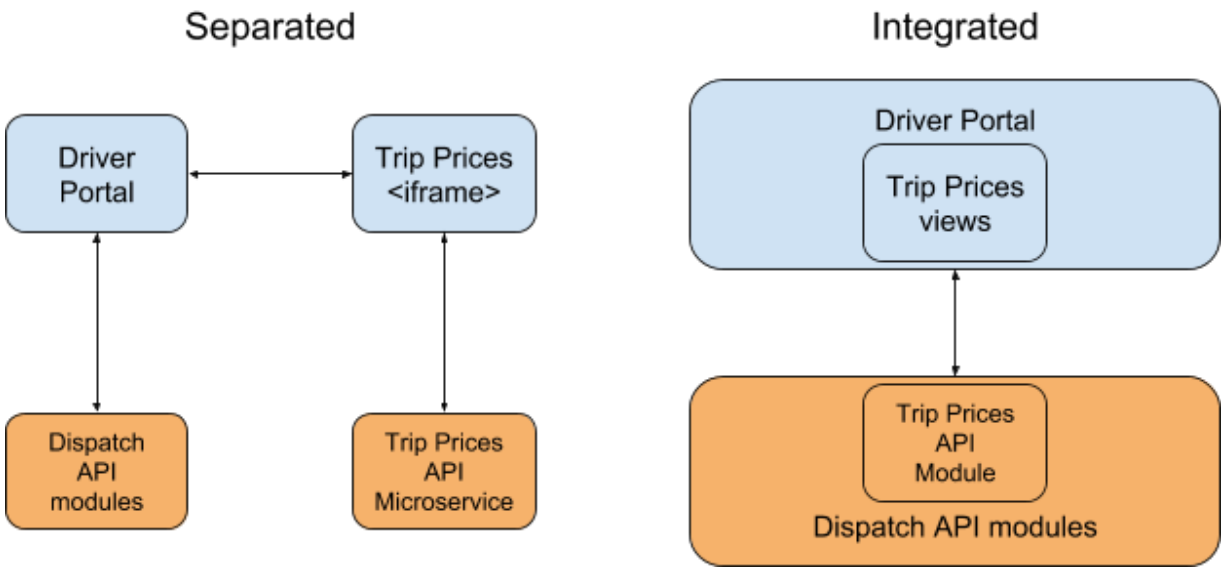


Image 3.3.2 - Separated and integrated frontend and or backend.

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 seperate 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:

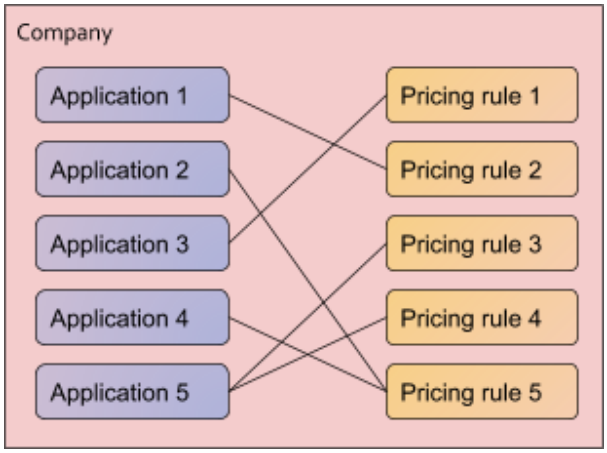


Image 4.1.1 - Company with applications and pricing rules.

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.

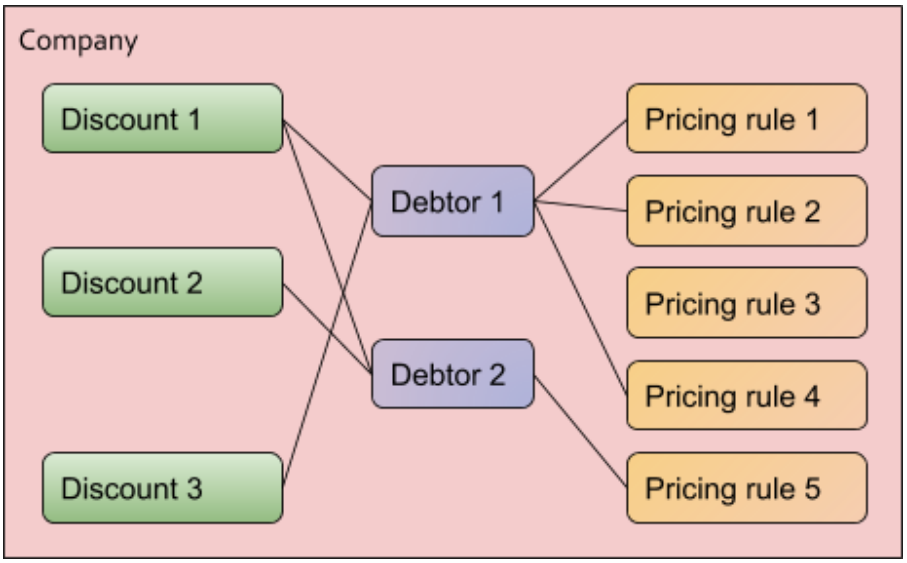


Image 4.2.1 - Debtors and the relation with pricing rules & discounts.

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 4.3.1 - 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 integrate the frontend in the existing portal

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.

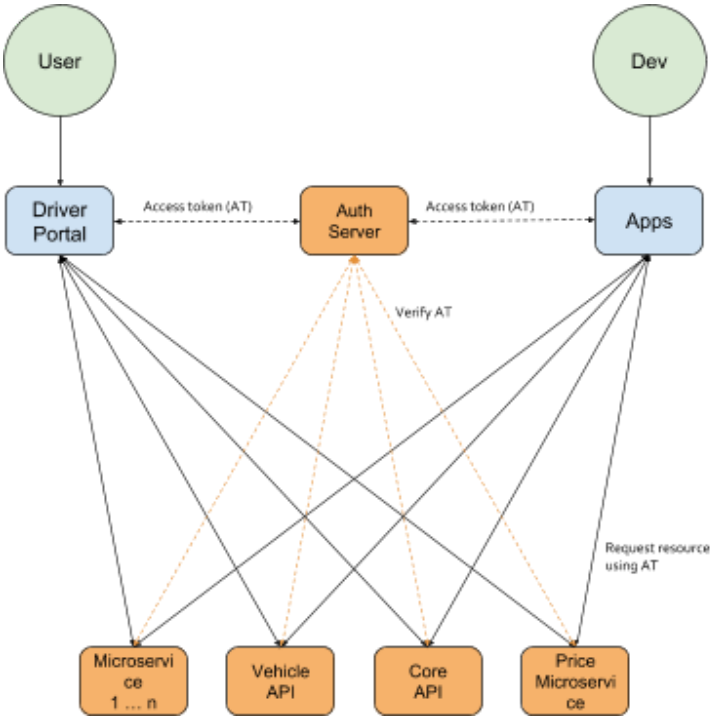


Image 4.4.1.1 - OAuth requests where tokens are verified by Auth Server.

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.

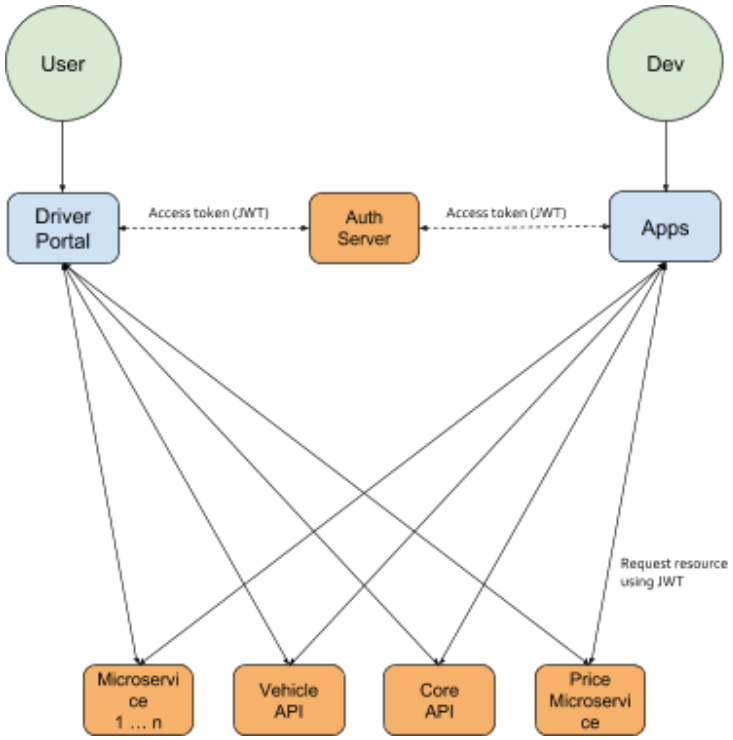


Image 4.4.2.1 - OAuth with stateless JWT token requests.

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.

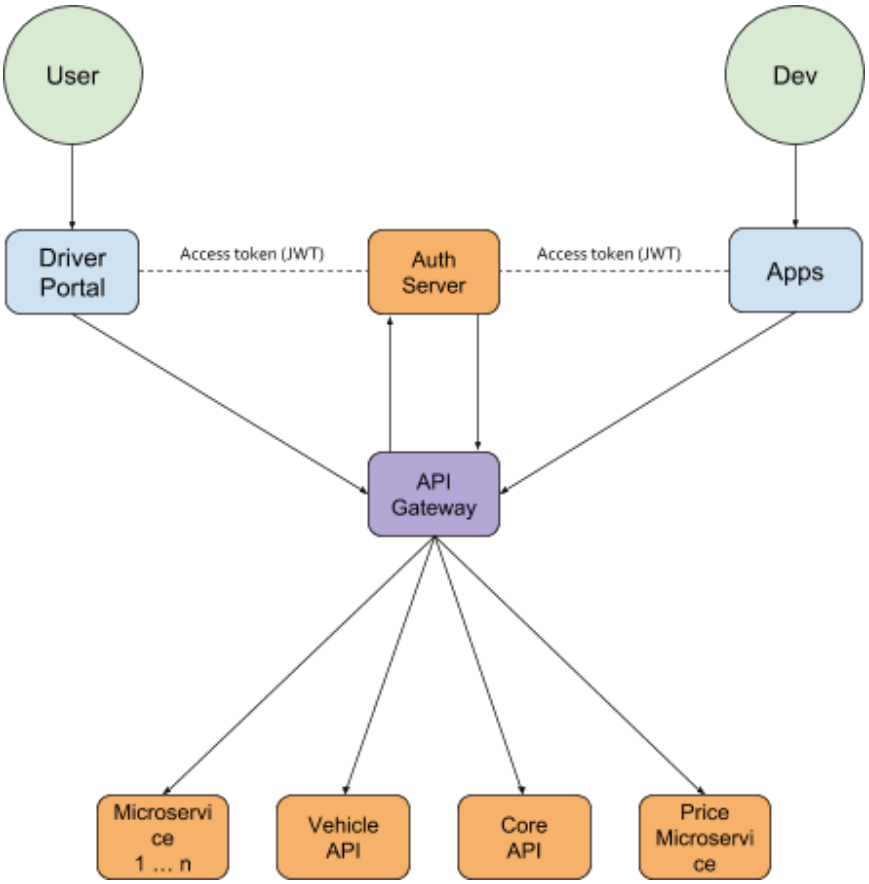


Image 4.4.3.1 - API Gateway.

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.

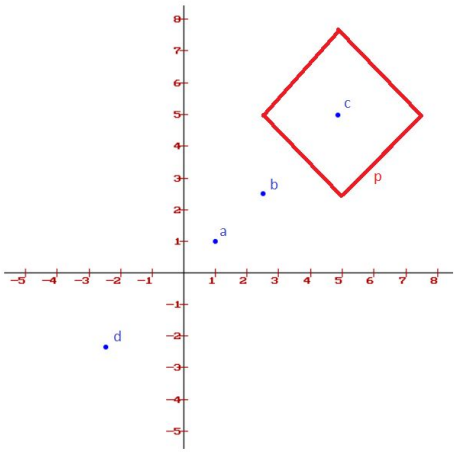


Image 4.5.1 - Four Points and one Polygon p containing Point c.

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.

```
// All points contained in polygon
SELECT ST_ASTEXT(POINT)
FROM POINT
WHERE
  ST_CONTAINS(
    (
        SELECT POLYGON
        FROM POLYGON
        WHERE id = 1
    ),
    POINT
  );
```

```
// All polygons containing point
SELECT ST_ASTEXT(POLYGON)
FROM POLYGON, POINT
WHERE
  POINT.id = 3 AND ST_CONTAINS(
        POLYGON.polygon,
        POINT.point
  )
```

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.

```
// 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]
            ]
        }
    }
})
```

```
// All polygons containing point
var p = db.point.findOne({ coordinates: [5, 5] })

db.polygon.find({
    shape: {
        $geoIntersects: {
            $geometry: {
                type: "Point",
                coordinates: [5, 5]
            }
        }
    }
})
```

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

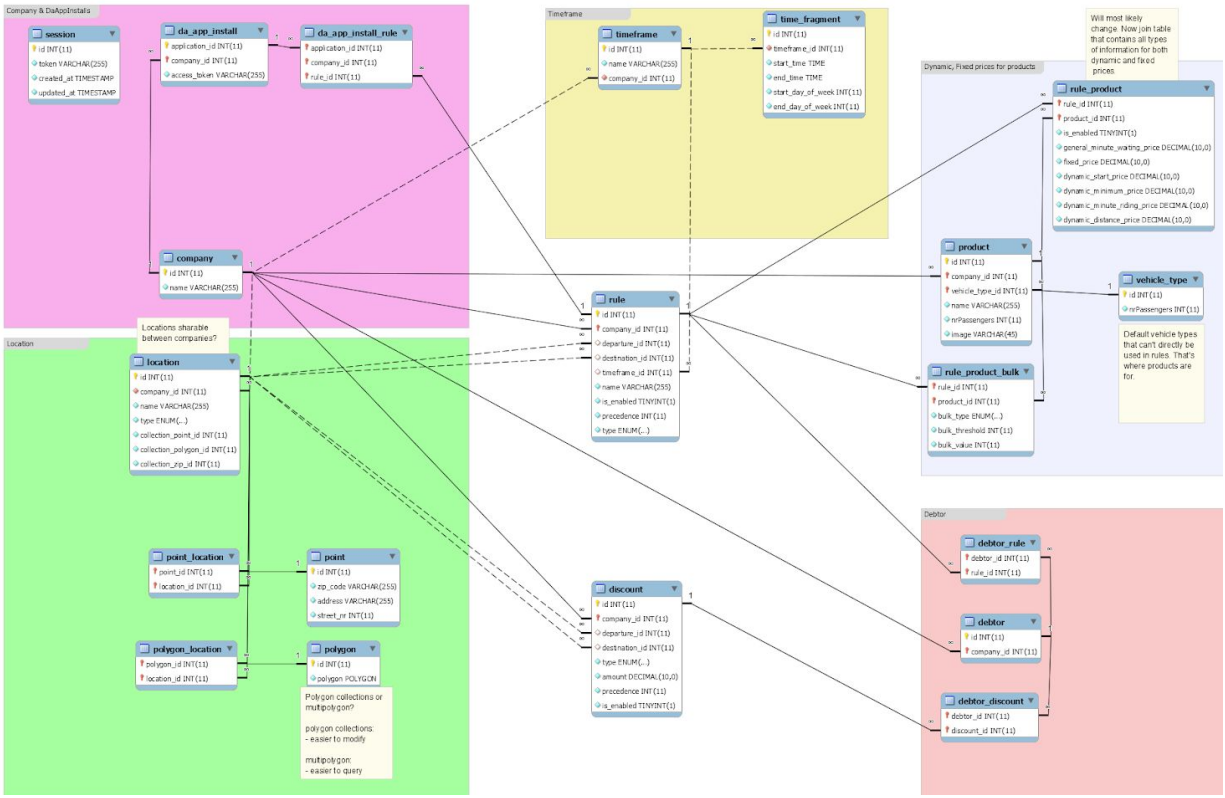


Image 4.7.1.1 - Rough schema for a relational database.

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: <ObjectId1>,
 user_token: "",
 settings: {
   is_begin_end_same_address: true
 }
}
```

```
// Rule document
// with embedded rule type
// with references to one discount (or many)
{
 _id: <ObjectId2>,
 application_id: <ObjectId1>,
 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. [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. [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_GeographyVSGeometry. [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. [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 Review and Proposal Slides

Online collection of slides: https://goo.gl/BR75HN

B.1 Sprint 1 - Dynamic Price Calculations

Sprint 1
Dynamic Price Calculations

Most complex but basic calculation using companyId, vehicleTypes, maxPassengers, enabled pricing rules ordered by precedence, trip distance and duration

Data Fixtures

Fake data is generated so that the process of developing the dynamic price calculation is consistent and swift across developer machines

Faking Data

Random data is inserted in the database using data fixtures

_id	name	maxPassengers	type	imagePath	companyId
5aa00f3ddd433...	Saloon	3	limo	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Estate	4	estate	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Bus	6	bus	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Minivan	6	minivan	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Limo	20	limo	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Granite purple car	9	limo	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Soft orange car	1	minivan	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Frozen violet car	2	saloon	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Plastic orchid car	8	minivan	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Plastic cyan car	7	limo	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Steel turquoise car	10	estate	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Soft azure car	9	estate	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Rubber silver car	6	estate	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Steel fuchsia car	5	bus	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Concrete white car	3	bus	https://goo.gl/TA...	5aa00f3ddd4337...

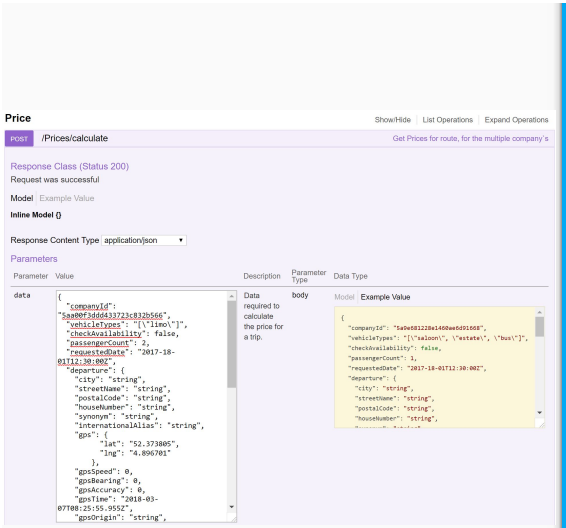
```
User: ...

Company: ...

Product:
  product{1..10}:
    maxPassengers: "...
    type: "{{random.arrayElement([
        \"saloon\",
        \"estate\",
        \"bus\",
        \"minivan\",
        \"limo\"
      ])}}"
    name: "{{commerce.productMaterial}} {{commerce.color}} car"

    imagePath: "https://goo.gl/TA829X"
    companyId: "@{company}"

PricingRule:
  pricingRule{1..10}:
    name: "{{commerce.productName}} Vehicle"
    isEnabled: "{{random.boolean}}"
    type: "{{random.arrayElement([
        \"dynamic\",
        \"fixed\"
      ])}}"
    precedence: "{{random.number}}"
    companyId: "@{company}"

ProductPricing:
  productPricing{1..100}:
    isEnabled: "{{random.boolean}}"
    minuteWaitingPrice: "0.25"
    fixedPrice: "0"
    dynamicStartPrice: "3.00"
    dynamicMinimumPrice: "5.00"
    dynamicMinutePrice: "0.32"
    dynamicDistancePrice: "2.22"
    pricingRuleId: "@{pricingRule.*}"
    productId: "@{product.*}"
```


Price Calculation

In this first sprint, the basic steps of a dynamic price calculation are orchestrated

Step 1 - PassengerApp sends request to TPS

The next couple of slides show the process of sending the request to our TPS service, and the way that our server processes the request before returning a response with a price calculation for each requested product



Request

As the documentation of the old system suggests, the query format in the yellow box is expected, and used as an example

Source: https://docs.dispatchapi.io/#get-prices-per-vehicle-type


```
{
     companyId:
     vehicleTypes:
     passengerCount:
     departure: {
          gps: {
               lat:
               lng:
          }
     destination: {
          gps: {
               lat:
               lng:
          }
     }
}
```

Data

The values on the left side of this slide are the only values that are currently being accepted by the endpoint

Step 2 - Obtaining ride distance and duration

The distance and duration of a trip are provided by the google directions API

The next slide shows the request parameters sent to google directions API, and the desired response attributes

Request

Fields used in google directions:

1. departure (gps: lat, lng)
2. destination (gps: lat, lng)

Response

Returned by google:

1. distance (in m)
2. duration (in s)

Step 3 - Querying our database for matches

While location matching is not part of the system yet, we could theoretically pass all the information we have at this moment to our database query to get the best possible match while ignoring the locations and timeframes for now

The query is performed for every vehicle type that the user wants to see, and returns exactly one best result for each

The next slide shows the request that would be sent by the Passenger App to our TPS microservice

Query

Fields used in query:
1. companyId
2. vehicleTypes
3. passengerCount

Fields unused:
4. departure
5. destination
6. pickupTime

Step 4 - Calculating the prices

After the query to the database has been made, the most complex work is done to calculate prices based on different rules provided and stored in our database by the group admins

A group admin can choose whether he would like the price to be calculated using tiers. He can flip a switch after he's defined the thresholds and tier prices for every one of his products

E.g. $0.5 dollar per km for the first 10 km, plus $0.4 * the next 10 km, plus $0.35 for the rest 2.54 km.
total = 5 + 4 + 0.889
final = max(9.89 + 3, 5)
final = 9.89

(this example only uses the distance metric)

total =

metric * metricPrice

or if tier pricing

each(threshold * thresholdPrice)

final = max(
total + startAmount,
minAmount
)

total: km * kmPrice

or

km - thresholds * kmPrice
+ (threshold * tierPrice)

Final: the price that is finally returned

Step 5 - Sending back the response

When all the prices have been calculated (for each vehicle type / product), the response is sent back to the PassengerApp

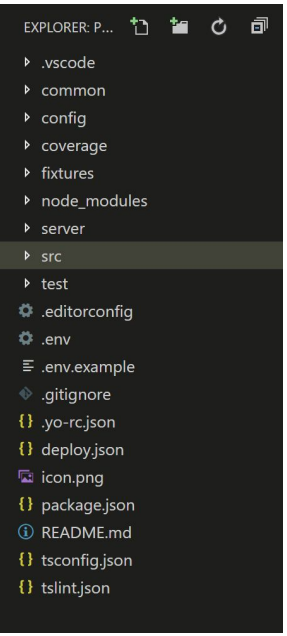
Response

Each vehicle type / product has a maximum of one result


```
[
  {
    "vehicleType": "saloon",
    "maxPassengers": 8,
    "fixedPrice": true,
    "price": {
      "currency": "EUR",
      "total": 1165,
      "breakdown": {
        "route": 1099,
        "tax": 66,
        "toll": 0,
        "parking": 0,
        "waiting": 0,
        "discount": 0
      }
    }
  },
  {
    "vehicleType": "limo",
    "maxPassengers": 5,
    "fixedPrice": true,
    "price": {
      "currency": "EUR",
      "total": 1165,
      "breakdown": {
        "route": 1099,
        "tax": 66,
        "toll": 0,
        "parking": 0,
        "waiting": 0,
        "discount": 0
      }
    }
  },
```


Project Structure

As Loopback 3 does not support Typescript out of the box, a separation between inherent loopback files and external functionality files is made, so that Typescript can be used for pieces of software that are decoupled from the framework



File Structure

common	Loopback models & schemas
config	Loopback config files
coverage	Test reporting
fixtures	Data fixtures for generating test data in db
server	Loopback server files
src	Typescript project
test	Typescript tests
.editorconfig .env .tsconfig .tslint	Space, tabs, line-ending styles Environmental variables Typescript settings Typescript linting

Tests

Tests are written using Mocha and Chai to guarantee that a functionalities continue to operate consistently, adhering to the FIRST mnemonic

1. Fast
2. Isolate
3. Repeatable
4. Self validating
5. Timely

Output

1. UNIT:
   Aims to test small units of code
2. INTEGRATION:
   Tests whether different parts of the system work together
3. Note:
   Current tests assume that the environment in which it resides is operational. For example: a google directions api key is set, the system is connected to the network, et cetera.

```
stefan@DESKTOP-M590E8U:/mnt/c/Projects/pricing-api$ yarn test
yarn run v1.5.1
$ tslint --fix src/**/*.ts{,x} --config tslint.json --project tsconfig.json
$ yarn run test:coverage
$ TS_NODE_COMPILER_OPTIONS='{"target":"es6"}' nyc --reporter=lcov yarn run test:unit
$ mocha -r ts-node/register "./test/**/*.spec.ts" --exit


  INTEGRATION: The .env file and environmental variables
    ✓ should load without throwing an error

  INTEGRATION: Server response status
    ✓ returns 200 on root page
    ✓ returns 404 everything else

  UNIT: GoogleDirections Settings
    ✓ can be mutated
    ✓ can accept an API key
    ✓ should detect invalid API key
    ✓ has API key set
    ✓ has travelMode defined

  INTEGRATION: Google API Service
    ✓ instantiation will succeed
    ✓ current environment has valid API key
    ✓ response to have { distance: 19.17, duration: 28.65 } (219ms)

  UNIT: PriceCalculation Class
    ✓ should throw an error on duplicate thresholds
    ✓ should have readonly taxPerc property
    ✓ should throw an error invalid pricing

  INTEGRATION: Price Calculation Different Cases
    ✓ should calculate a price without thresholds
    ✓ calculates price with distance threshold
    ✓ calculates price with duration threshold
    ✓ has a recursive function to calculate cascading thresholds
    ✓ calculates price with distance and duration thresholds
    ✓ should calculate a price with cascaded duration thresholds


  20 passing (360ms)

Done in 9.98s.
stefan@DESKTOP-M590E8U:/mnt/c/Projects/pricing-api$
```


Debugging

Set the debug flag to true to display errors and logs during the tests

```
import debug from '../../debug';

debug(true);

describe('UNIT: PriceCalculation Class', () => {

 it('should throw an error on duplicate...
```

```
4 passing (93ms)
5 failing

  1) INTEGRATION: Price Calculation Different Cases
       should calculate a price without thresholds:

      AssertionError: expected { Object (vehicleType ...

      + expected - actual

       {
      -  "fixedPrice": false
      -  "maxPassengers": 3
      -  "price": {
      -    "breakdown": {
      -      "discount": 0
      -      "parking": 0
      -      "route": 83
      -      "tax": 5
      -      "toll": 0
      -      "waiting": 0
      -    }
      -    "currency": "EUR"
      -    "total": 88
      -  }
      -  "vehicleType": "saloon"
      +  "discount": 0
      +  "parking": 0
      +  "route": 83
      +  "tax": 5
      +  "toll": 0
      +  "waiting": 0
       }
```

Tests: coverage reporting

All files

84.88% Statements 146/172 **72%** Branches 36/50 **87.18%** Functions 34/39 **85.37%** Lines 140/164

File	Statements		Branches		Functions		Lines	
src	76.47%	26/34	44.44%	4/9	60%	3/5	80.65%	25/31
src/boot	100%	4/4	100%	0/0	100%	1/1	100%	4/4
src/services/directions	75.44%	43/57	69.57%	16/23	93.75%	15/16	74.55%	41/55
src/services/prices	94.81%	73/77	88.89%	16/18	88.24%	15/17	94.59%	70/74

Istanbul tests checks to see what lines of code were run. The report shows useful information to improve the test coverage of a project.

```
14      /**
15       * Start price calculations. The distance and duration metrics
16       * are fetched by the directionsService using an async function
17       * before calculate is used to calculate the trip price.
18       */
19      public async breakdown(pricing: pricing): Promise<object> {
20
21      culator.validPricingOrError(pricing);
22      trics = await this.directionsService.directions();
23  5x      if (!metrics) {
24          throw new HttpError('Metrics not provided for price calculation.');
25          }
26
27  5x      const routePrice = this.calculate(pricing, <metrics>metrics);
28  5x      const taxPrice = PriceCalculator.taxPerc * routePrice;
29  5x      const tollPrice = 0; // @todo
30  5x      const parkingPrice = 0; // @todo
31  5x      const waitPrice = pricing.prices.minuteWaitingPrice * 0; // @todo
32  5x      const discountPrice = 0; // @todo
```

if path not taken

B.2 Sprint 2 - Breakdown Proposal

Price Breakdown Proposal

Including and excluding VAT

Property types

- number
- hidden
- computed
- total

A number is a fixed amount that has been calculated

Hidden is not included in the price calculation breakdown, but is included to demonstrate sub-calculations

Computed is a price calculated using the numbers in the breakdown

Total is the aggregated price based on computed and number properties

Requirements

- The breakdown contains computed and numbers
- VAT is included in all the properties
- Total is the sum of all the properties in the breakdown

In the future, waiting, toll, and parking properties may have their own VAT percentage

In the future, waiting, toll, and parking properties are included in the breakdown, therefore we will assume that they are provided

Including VAT

The following slides present the calculation assuming that VAT is included

```
"price": {
  "breakdown": {
    "discount":
    "parking":
    "route":
    "toll":
    "waiting":
  }
  "currency": "EUR"
  "total":
  "tax": {
      amount:
      percentage: 6
  }
}
```

The breakdown, total and currency are properties of the price

As said before, the total must aggregate all properties in the breakdown

The currency is EUR


```
"price": {
  "breakdown": {
    "discount":
    "parking": 2
    "route": 65
    "toll": 5
    "waiting": 2.8
  }
  "currency": "EUR"
  "total":
  "tax": {
      amount:
      percentage: 6
  }
}
```

Steps:

1. Parking, toll, waiting and route prices are calculated, tax percentage is added


```
"price": {
  "breakdown": {
    "discount": -11.22
    "parking": 2
    "route": 65
    "toll": 5
    "waiting": 2.8
  }
  "currency": "EUR"
  "total":
  "tax": {
      amount:
      percentage: 6
  }
}
```

Steps:

1. Parking, toll, waiting and route prices are calculated, tax percentage is added
2. Discount
   a. Is stated as a positive or negative fixed amount ✔ -11.22
   b. Is negative or positive percentage of subtotal ✔ (-15%) 74.8*-0.15 = -11.22


```
"price": {
  "breakdown": {
    "discount": -11.22
    "parking": 2
    "route": 65
    "toll": 5
    "waiting": 2.8
  }
  "currency": "EUR"
  "total": 63.58
  "tax": {
      amount:
      percentage: 6
  }
}
```

Steps:

1. Parking, toll, waiting and route prices are calculated, tax percentage is added
2. Discount
   a. Is stated as a positive or negative fixed amount
   b. Is negative or positive percentage of subtotal
3. Total is added ✔ 63.58

```
"price": {
  "breakdown": {
    "discount": -11.22
    "parking": 2
    "route": 65
    "toll": 5
    "waiting": 2.8
  }
  "currency": "EUR"
  "total": 63.58
  "tax": {
      amount: 3.6
      percentage: 6
  }
}
```

Steps:

1. Parking, toll, waiting and route prices are calculated, tax percentage is added
2. Discount
   a. Is stated as a positive or negative fixed amount
   b. Is negative or positive percentage of subtotal
3. Total is added
4. Tax calculated ✔ `63.58 / (100 + tax.percentage) * tax.percentage = 3.6`

Excluding VAT

The following slides provide a calculation assuming VAT is excluded


```
"price": {
  "breakdown": {
    "discount":
    "parking":
    "route":
    "tax":
    "toll":
    "waiting":
  }
  "currency": "EUR"
  "total":
}
```

The breakdown, total and currency are properties of the price

As said before, the total must aggregate all properties in the breakdown

The currency is EUR


```
"price": {
  "breakdown": {
    "discount":
    "parking": 2
    "route": 65
    "tax":
    "toll": 5
    "waiting": 2.8
    "subtotal":
  }
  "currency": "EUR"
  "total":
}
```

Steps:

1. Parking, toll, waiting and route prices are calculated

```
"price": {
  "breakdown": {
    "discount":
    "parking": 2
    "route": 65
    "tax":
    "toll": 5
    "waiting": 2.8
    "subtotal": 74.8
  }
  "currency": "EUR"
  "total":
}
```

Steps:

1. Parking, toll, waiting and route prices are calculated
2. A subtotal = parking + route + toll + waiting, is calculated

✔ 65+5+2+2.8 = 74.8


```
"price": {
  "breakdown": {
    "discount": -11.22
    "parking": 2
    "route": 65
    "tax":
    "toll": 5
    "waiting": 2.8
    "subtotal": 74.8
  }
  "currency": "EUR"
  "total":
}
```

Steps:

1. Parking, toll, waiting and route prices are calculated
2. A subtotal = parking + route + toll + waiting, is calculated
3. Discount..
   a. Is stated as a positive or negative fixed amount ✔ -11.22
   b. Is negative or positive percentage of subtotal ✔ (-15%) 74.8*-0.15 = -11.22


```
"price": {
  "breakdown": {
    "discount": -11.22
    "parking": 2
    "route": 65
    "tax": 3.8148
    "toll": 5
    "waiting": 2.8
    "subtotal": 74.8
  }
  "currency": "EUR"
  "total":
}
```

Steps:

1. Parking, toll, waiting and route prices are calculated
2. A subtotal = parking + route + toll + waiting, is calculated
3. Discount..
   a. Is stated as a positive or negative fixed amount
   b. Is negative or positive percentage of subtotal
4. Tax = (discount + subtotal) * tax_percentage

✔ (6%) (-11.22+74.8)*0.06 = 3.8148


```
"price": {
  "breakdown": {
    "discount": -11.22
    "parking": 2
    "route": 65
    "tax": 3.8148
    "toll": 5
    "waiting": 2.8
    "subtotal": 74.8
  }
  "currency": "EUR"
  "total": 67.3948
}
```

Steps:

1. Parking, toll, waiting and route prices are calculated
2. A subtotal = parking + route + toll + waiting, is calculated
3. Discount..
   a. Is stated as a positive or negative fixed amount
   b. Is negative or positive percentage of subtotal
4. Tax = (discount + subtotal) * tax_percentage
5. Total = discount + subtotal + tax

✔ -11.22+74.8 = 63.58
+3.8148 = 67.3948

```
"price": {
  "breakdown": {
    "discount": -11.22
    "parking": 2
    "route": 65
    "tax": 3.8148
    "toll": 5
    "waiting": 2.8
    "subtotal": 74.8
  }
  "currency": "EUR"
  "total": 67.5
}
```

Steps:

1. Parking, toll, waiting and route prices are calculated
2. A subtotal = parking + route + toll + waiting, is calculated
3. Discount..
   a. Is stated as a positive or negative fixed amount
   b. Is negative or positive percentage of subtotal
4. Tax = (discount + subtotal) * tax_percentage
5. Total = discount + subtotal + tax
6. Round total price to half a decimal

B.3 Sprint 2 - Authentication Proposal

Authentication and Identity Management of the Microservice Proposal

Taking or delegating responsibility of authentication within the system architecture

Definition of a Microservice

- Small service decomposed from a monolith
- Isolated and independently deployable
- Stateless and less fragile when changes are introduced
- Single responsibility
- Advice was provided in the Phase I - Pregame document

Aspects of Authentication and Identity Management

Name	Explanation	Example
- Responsibility	System concerned of authenticating users?	Core, external service or microservice itself
- Locality	Where is user data stored?	A single database, all databases
- Authorization	How do we identify user and roles?	CompanyId, DaAppInstall ...
- Statefulness	How is the state of authentication shared between services?	In the request, in shared or separate sessions

Aspects of Authentication and Identity Management

Name	Explanation	Example
Responsibility	System concerned of authenticating users	Core system, external service or the microservice itself
Locality	Where user data stored	One single database, every database, a separate database
Authorization	How users and roles are identified	CompanyId, DaAppInstallId, a separate combined id
Statefulness	How the user state is synchronized between services	In the request, in shared or separate sessions, via a token

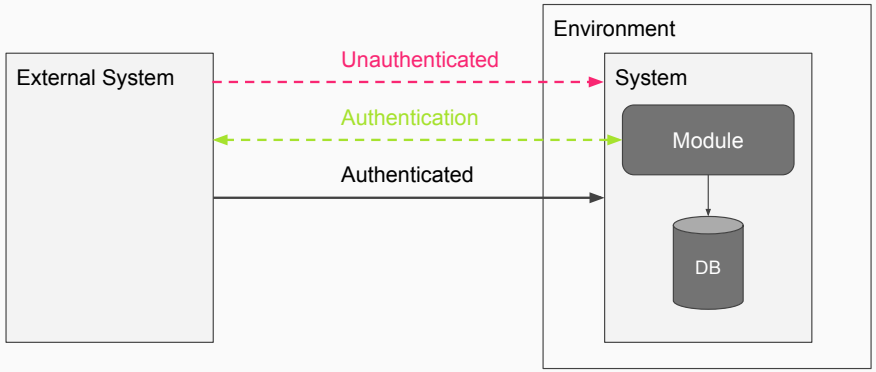
Examples

Here are four examples 1 … 4 increasing from basic to more extreme implementations.

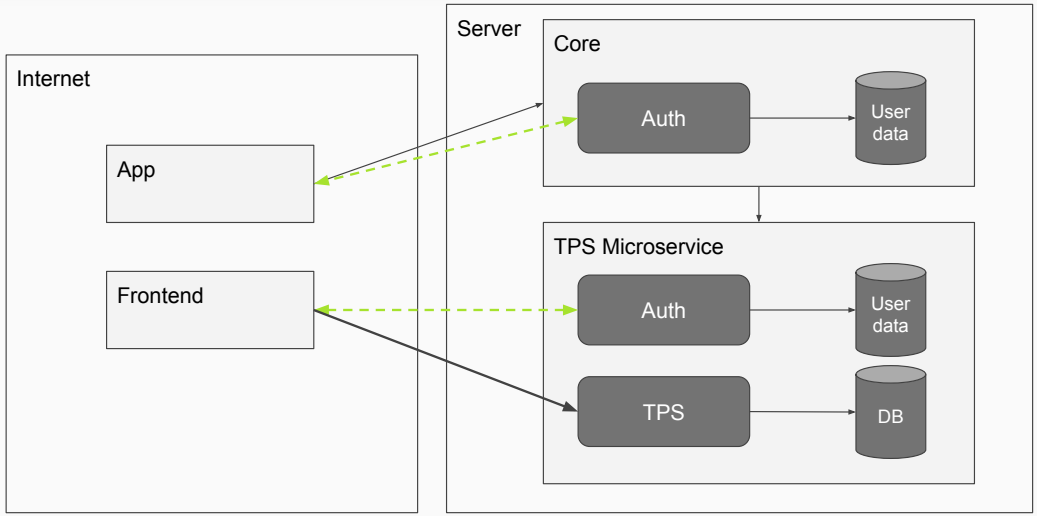
The four aspects are discussed after each figure:

- Responsibility
- Locality
- Authorization
- Statefulness

Symbols used in the Examples



Ex 1



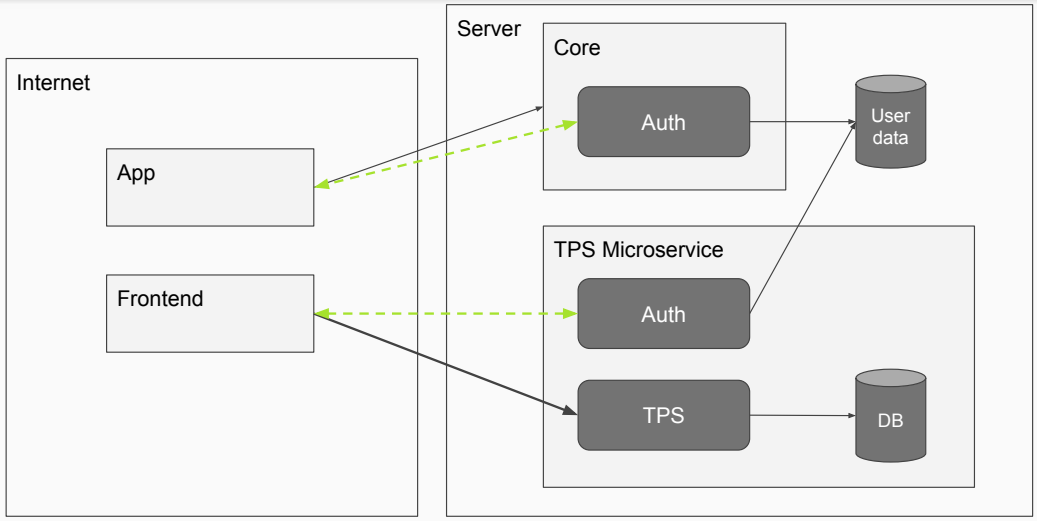
Ex 1 - Aspects

The TPS microservice authenticates its users. It has its own database with user and company data.

Authorization is handled by checking the user data in the database. Sessions are handled by the microservice. So the state resides in the microservice.

The microservice is totally independent, except for the fact that the data that is mutated in other systems must be synchronized in some fashion.

Ex 2

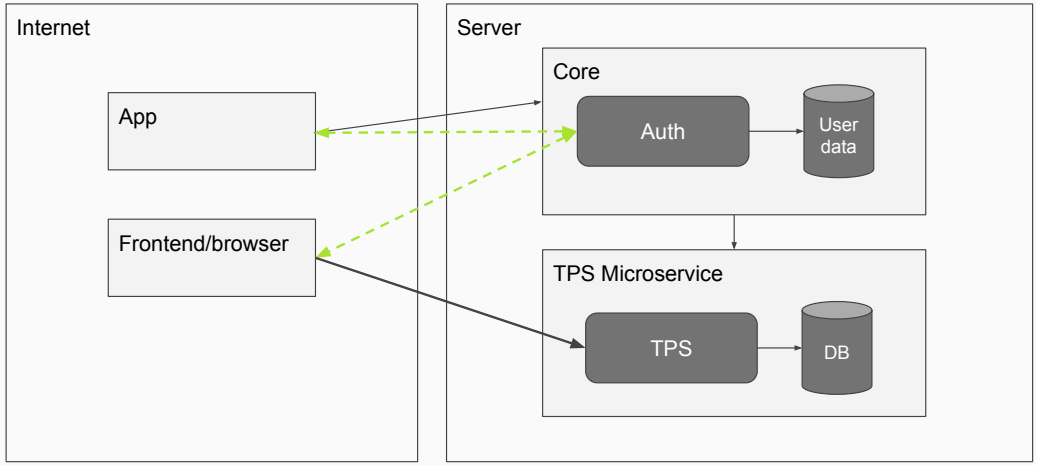


Ex 2 - Aspects

Like example 1, authentication is handled by the microservice. Only this time it connects directly with a database that stores user information.

Depending on how sessions are handled, the state can be shared amongst systems. But when other systems are required to make use of the microservice, more and more sources that contain the state of users need to be shared with the microservice.

Ex 3



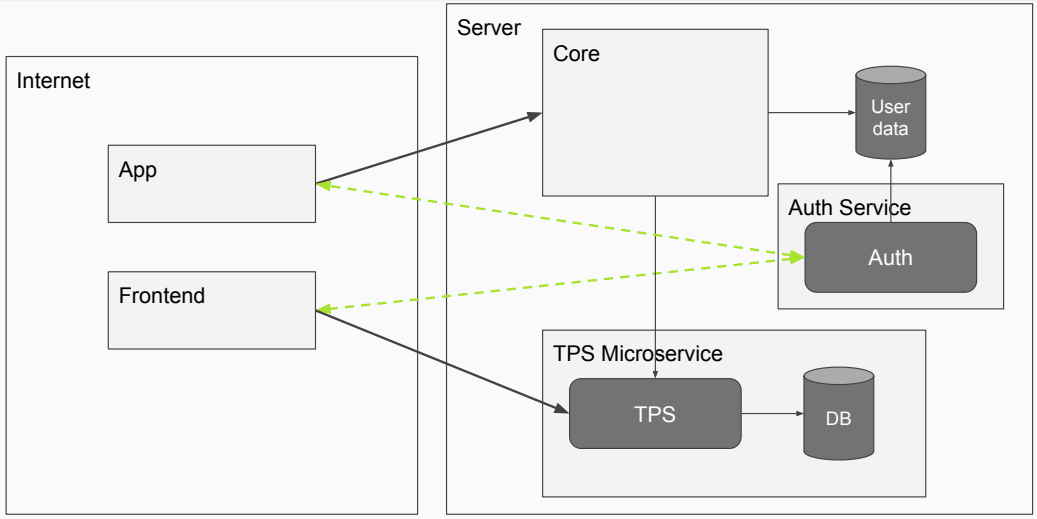
Ex 3 - Aspects

In example 3, the Core system is the only system able to provide authentication tokens. This token must be used to transfer authorization and identity information to the microservice in a stateless manner, because the microservice has no concept of the state of authentication.

A JWT can be used to transfer state in this case.

If more systems have to make use of the microservice in the future, they depend on the core system anyhow.

Ex 4



Ex 4 - Aspects

In this example, future systems don't depend on the core system. But the core system does depend on the Authentication service in order to make use of other microservices.

Concluding

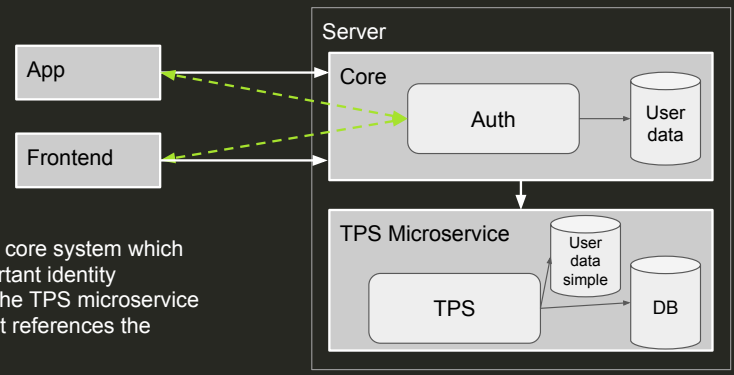
- Managing invalidation of JWT's
  - Invalidating individual tokens conditionally
  - https://stormpath.com/blog/token-auth-spa
- Keeping payloads up-to-date when data changes
  - User switches from company, therefore the frontend must act!

B.4 Sprint 2 - Authentication and Authorization

Sprint 2 Authentication and Authorization

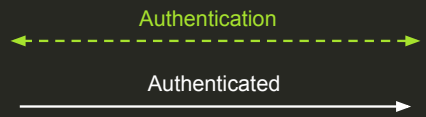
Authentication, VAT, Discounts, Improved Breakdown, Cascading Threshold Calculations, and refactors

Selected Authentication Proposal



Applications communicate with the core system which provides a JWT that includes important identity information in the token payload. The TPS microservice has a simple concept of a User that references the DaAppInstall of a company.

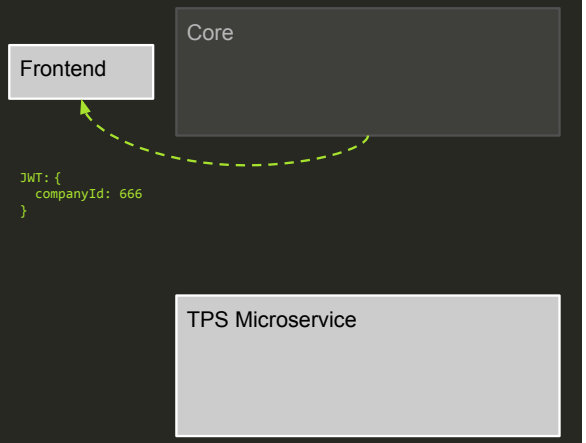
The Microservice is capable of decrypting the JWT revealing the identity details to find the user stored in the database, so that the correct rules linked to a DaAppInstall are used for the price calculation.



Authentication

Core signs JWT that has the identity of the user contained in the payload, e.g.:

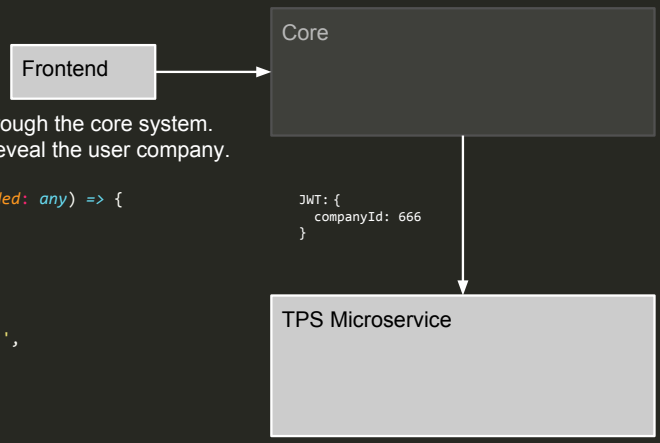
```
const cert = process.env.JWT_SECRET;
const HOURS_24 = 86400;

jwt.sign({ companyId: '666' }, cert, {
  expiresIn: HOURS_24,
  algorithm: 'HS256'
}, (err, token) => {
  if (err) return cb(err);
  return cb(null, token);
});
```



Authentication

The frontend makes a request directed at the Microservice, which is tunneled through the core system. The Microservice decrypts the token to reveal the user company.

```
jwt.verify(token, cert, (err: Error, decoded: any) => {
  if (err) {
    return res.status(500)
      .send({
        auth: false,
        message: 'Invalid token provided.',
      });
  }
  console.info(decoded);
  // { companyId: 666, iat: 1521552244, exp: 1521638644 }
  next();
});
```



Authentication

Middleware:

1. Fetches token
2. Fetches secret
3. Verifies token with secret or public key
4. Adds credentials to req
5. next( )

```
/**
 * Authentication middleware.
 */
export const auth = (req: Request, res: Response, next: NextFunction) => {

 // Don't use this cert in any of the warning responses, it's secret
 const cert = process.env.JWT_SECRET;
 const JWT_HEADER = process.env.JWT_HEADER || 'x-access-token';
 const token = req.headers[JWT_HEADER] : req.query.access_token;

 // It can be safe under the following circumstances:
 // 1. the JWT is one-time time usage only
 // 2. the jti and exp claims are present in the token
 // 3. the receiver properly implements replay protection jti and exp
 if (process.env.NODE_ENV === 'development' && req.query.access_token) {
   return warn(res, 449, `Don't send tokens via an URL (preferably)`);
 }

 // No token, no access
 if (!token) return warn(res, 403, 'No token provided.');

 // No secret found, still no access
 if (!cert) return warn(res, 403, 'Authentication unavailable.');

 // Verify token and add to request
 jwt.verify(token, cert, (err: Error, decoded: any) => {
   if (err) return warn(res, 403, 'Invalid token provided.');
   addCredentialsToReq(req, decoded);
   next();
 });
};
```


Authentication

1. Changes must be made to the core system
2. All communications will be held through the core system
3. Secrets must be known to both core and microservice
4. Secrets should be stored in the .env file
5. Secrets should be updated regularly

Breakdown Proposal

```
"price": {
  "breakdown": {
    "discount": -11.22
    "parking": 2
    "route": 65
    "toll": 5
    "waiting": 2.8
  }
  "currency": "EUR"
  "total": 63.58
  "tax": {
      "amount": 3.6
      "percentage": 6
   }
}
```

These values are a sum of the total

Breakdown Proposal

```
"price": {
  "breakdown": {
    "discount": -11.22
    "parking": 2
    "route": 65
    "toll": 5
    "waiting": 2.8
  }
  "currency": "EUR"
  "total": 63.58
  "tax": {
      "amount": 3.6
      "percentage": 6
   }
}
```

The tax is calculated but is part of the total, as VAT is included

Improved Breakdown

1. Total is the sum of the breakdown
2. Tax is included in the total, and thus included in the breakdown prices
3. Tax is based on country default tax percentage
4. The discussion is found in [the breakdown proposal slides]

VAT

```
// copy.country
country:
   { name: 'Netherlands',
     code: 'NL',
     defaultTax: 6,
     defaultCurrency: 'EUR' },
```

```
instance.breakdown(copy)
        .then((data: Response) => {
          expect(data)
            .deep.equal({
              price: {
                breakdown: {
                  route: 83,
                  toll: 0,
                  parking: 0,
                  waiting: 0,
                  discount: 0,
                },
                tax: {
                  amount: 4.5,
                  // 4.7 = 83 / 106 * 6
                  percentage: 6,
                },
                currency: 'EUR',
                // 66.32 = 82.83 - 16.5
                total: 66.5,
              }
            });
```


VAT

1. Tax is calculated back from included VAT prices
2. Tax percentages differ per country
3. More details in [the price breakdown proposal]

Established formula:

```
[price / (100 + tax.percentage) * tax.percentage]
```


Discounts

```
// copy.discount
discount:
  { name: 'Discount percentage test',
    value: -20
    isEnabled: true,
    type: 'percentage',
    precedence: 88547,
    companyId: 5aa1585990e4d72312f882db }
```

```
instance.breakdown(copy)
        .then((data: Response) => {
          expect(data)
            .deep.equal({
              price: {
                breakdown: {
                  route: 83,
                  toll: 0,
                  parking: 0,
                  waiting: 0,
                  // -16.6 = -.2 * 83
                  discount: -16.5,
                },
                ...
                currency: 'EUR',
                // 66.32 = 82.83 - 16.5
                total: 66.5,
              }
            });
```

Discounts

1. A discount can be negative or positive
2. A discount can be disabled or enabled
3. A discount can be a fixed amount or percentage
4. A discount is calculated and is part of the breakdown total
5. Discounts are not constrained by location or timeframes yet

Cascading Threshold Calculations

```
// Calculate total price
const result = Price.total(
  {
    distance: 45,
    duration: 25,
  },
  [
    {
      type: 'duration',
      threshold: 19.5,  // price after 20 min is 0.45
      value: 0.15,
    },
    {
      type: 'duration',
      threshold: 15,    // price after 15 min is 0.25
      value: 0.25,
    },
    {
      type: 'duration',
      threshold: 12,    // price after 10 min is 0.15
      value: 0.45,
    },
    {
      type: 'distance',
      threshold: 20,    // price after 20 km is 0.35
      value: 0.75,
    },
  ...
```

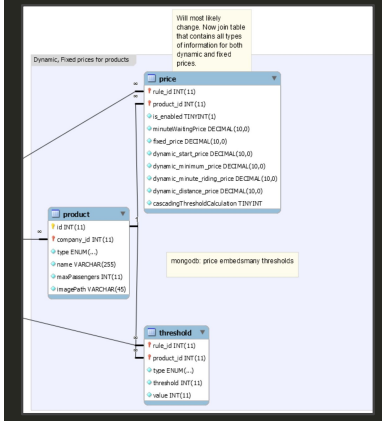
```
...
    {
      type: 'distance',
      threshold: 15,    // price after 15 km is 0.25
      value: 0.90,
    },
    {
      type: 'distance',
      threshold: 10,    // price after 10 km is 0.10
      value: 1.25,
    },
  ],
  {
    distance: calculation(1.5),
    // Distance is 45 km
    // The first threshold is found at 10 km
    // So the first 10 km cost 1.5 per km
    // The next 5 up until 15 km cost 1.25 per km
    // The next 5 up until 20 km cost 0.90 per km
    // The final 25 km cost 0.75 per km
    //
    // Distance price is 44.5
...
```

```
...
    duration: calculation(0.4),
    // Duration is 25 min
    // The first threshold is found at 10 min
    // So the first 12 min cost 0.4 per min
    // The next 3 up until 15 min cost 0.45 per min
    // The next 4.5 up until 19.5 min cost 0.25 per min
    // The final 5.5 min cost 0.15 per min
    //
    // Duration price is 8.1
  },
  true,
);
// Therefore result is 52.6 = 44.5 + 8.1
expect(result).to.equal(52.6);


// A function to generate a calc func with default price
const calculation = (defaultPrice: number) =>
  (price: number = defaultPrice, metric: number) =>
    price * metric;
```


Cascading Threshold Calculations

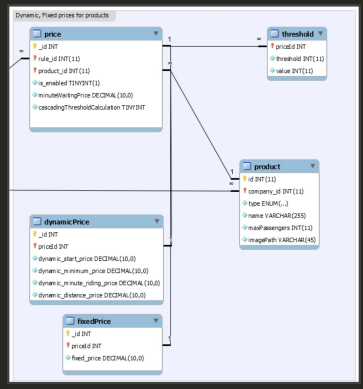
1. Either no thresholds have been provided, the normal calculation function is called straight away.
2. If thresholds are provided, there are two options:
   a. Cascade option is true
   b. Cascade option is false
3. If cascade is false, the price of the last threshold that has been surpassed will be used to calculate the price per metric.
4. If cascade is true, the first couple of km say, will be calculated with the normal price. The next km's will be calculated using the first surpassed threshold, the next km's with the next threshold price …

Refactors: DB Schema



Old:
Contained pricing information of all types in one table.

New:
Separated information into individual tables while retaining core information in original price table.



Refactors: Aggregate

```
// util.ts
const exhaustList = (array, func, next) => {
  if (array.length < 1) return next(array);
  const pop = array.pop();
  func(pop, array, next);
};

// common/price.js
const queryPoppedVehicle = (pop, array, next) => {
  aggregateQuery(pop, (result) => {
    exhaustList(array, queryPoppedVehicle, (newArray) => {
      if (result[0]) newArray.push(result[0]);
      return next(newArray);
    })
  })
}
...
exhaustList(vehicleTypes, queryPoppedVehicle, (rules) => {
  const promises = rules.map(rule => {
    return priceCalculator.breakdown(rule);
  });
  Promise.all(promises)
    .then(breakdowns => {
      cb(null, breakdowns);
    })
    .catch(error => {
      cb(error);
    })
})
```

Old:

Executed query for every vehicle type in the vehicleTypes array recursively, limit 1.

New:

Executes one aggregate.

```
// common/price.js
aggregateQuery(vehicleTypes, (rules) => {
  const promises = rules.map(rule => {
    return priceCalculator.breakdown(rule);
  });
  Promise.all(promises)
    .then(breakdowns => {
      cb(null, breakdowns);
    })
    .catch(error => {
      cb(error);
    })
});
```


Refactors

1. DB Schema
   a. Split the price table into three tables
   b. `price` is the junction table
   c. `fixedPrice` is the fixed price table
   d. `dynamicPrice` is the dynamic price table
   e. and
2. Aggregate
   a. Now performs 1 query instead of sum(vehicleTypes) queries

Flow

1. Price.calculate endpoint is called
2. JWT token payload is decrypted
3. Directions service instance created
   a. Immediately fetches distance & duration
4. Price calculator instance created
   a. Directions service is passed async
5. Aggregate query is created taking vehicleTypes from body & companyId from JWT payload
6. The query is performed and the resulting pricing rules are mapped to the Price calculator instance
7. The instance calculates:
   a. totalPrice (routePrice + tollPrice + parkingPrice + waitPrice + discountPrice)
   b. priceVAT (% VAT of totalPrice)The Price calculator promises a calculation for each pricing rule (per vehicle type)
8. If a promise fails, an empty array is returned
9. Else all breakdowns are returned in an array

Flow: Breakdown Method

```
/**
 * Start price calculations. The distance and duration metrics
 * are fetched by the directionsService using an async function
 * before calculate is used to calculate the trip price.
 */
public async breakdown(pricing: pricing): Promise<breakdown> {

  Price.validPricingOrError(pricing);
  const metrics = await this.directionsService.directions();
  if (!metrics || metrics.distance < 0 || metrics.duration < 0) {
    return Promise.reject('Metrics not valid for price calculation.');
  }

  const parkingPrice = 0;
  const routePrice = Price.calculators[pricing.rules.type](pricing, metrics);
  const tollPrice = 0;
  const waitPrice = pricing.prices.minuteWaitingPrice * 0;
  const discountPrice = pricing.discount
    ? pricing.discount.type === 'percentage'
      ? percentOf(pricing.discount.value, routePrice)
      : pricing.discount.value
    : 0;

  ...
```

```
...
  const vatPerc = pricing.country.defaultTax;
  const totalPrice = Math.max(0, routePrice
    + tollPrice
    + parkingPrice
    + waitPrice
    + discountPrice);
  const { priceExVAT, priceVAT } = excludeVatOf(vatPerc, totalPrice);

  return Promise.resolve({
    vehicleType: <vehicleType>pricing.type,
    maxPassengers: pricing.maxPassengers,
    price: {
      breakdown: {
        route: roundHalfDecimal(routePrice),
        toll: roundHalfDecimal(tollPrice),
        parking: roundHalfDecimal(parkingPrice),
        waiting: roundHalfDecimal(waitPrice),
        discount: roundHalfDecimal(discountPrice),
      },
      currency: pricing.country.defaultCurrency,
      total: roundHalfDecimal(totalPrice),
      tax: {
        amount: roundHalfDecimal(priceVAT),
        percentage: vatPerc,
      },
    },
  });
}
```

B.5 Sprint 3 - Products and Pricing

Sprint 3
Products and Pricing

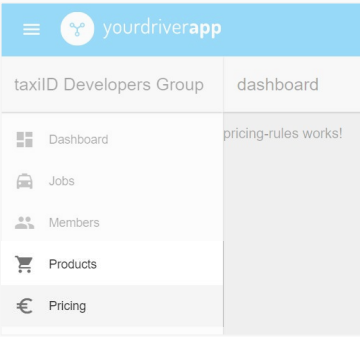
Products and pricing rules, automatically managing the many-to-many relations

Views

During this sprint, the portal project was set up for development, and two categories were added to the group sidebar:

1. Products
2. Pricing

The following slides show the views that have been implemented



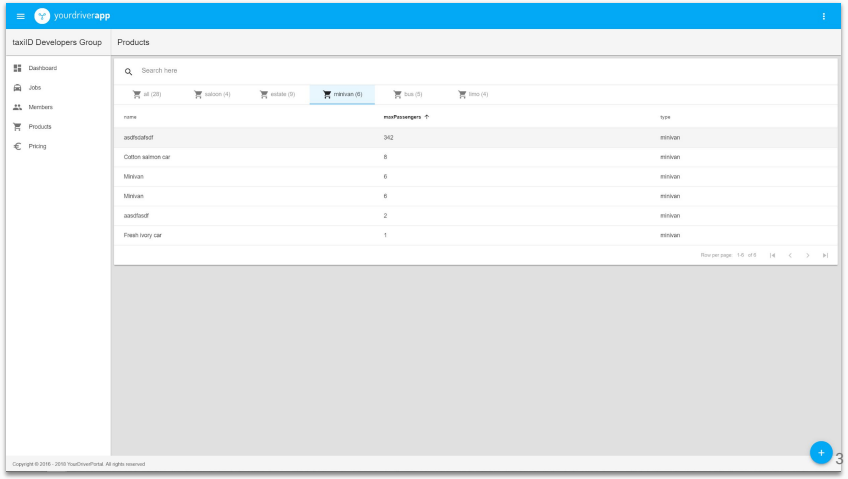
Products Overview

The products overview shows products per vehicle type

Products can be filtered and sorted

Clicking on a products will bring the user to the detail page

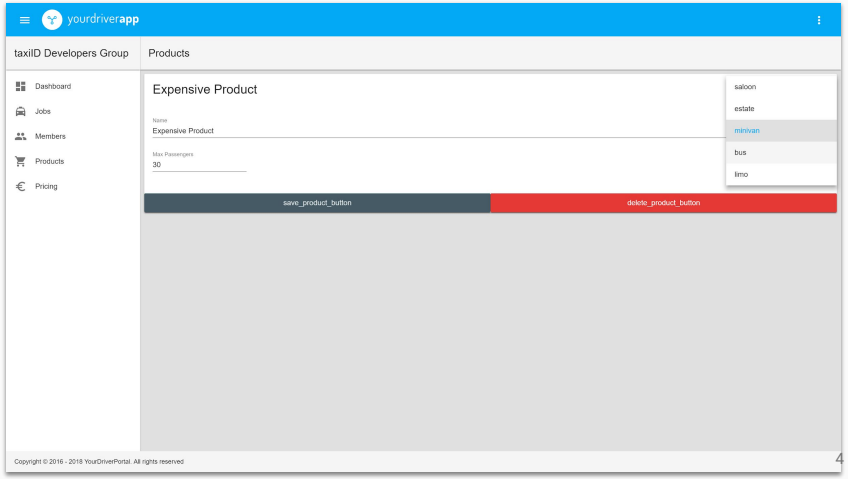
The button in the bottom right corner creates a new vehicle that can be edited and saved in the detail page



Products Detail Page

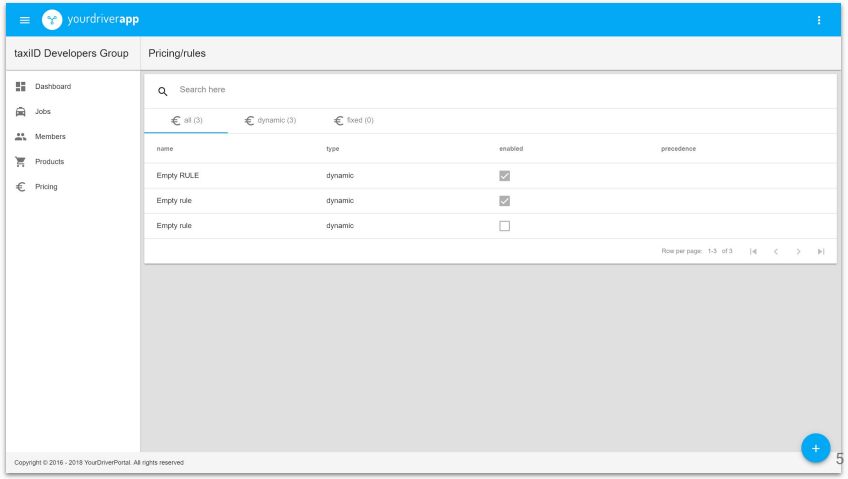
The products detail page shows the properties that belong to the product

The product may be deleted, upon which a confirmation screen protects the user against accidentals



Pricing Overview

The pricing overview has similar characteristics to that of the products overview

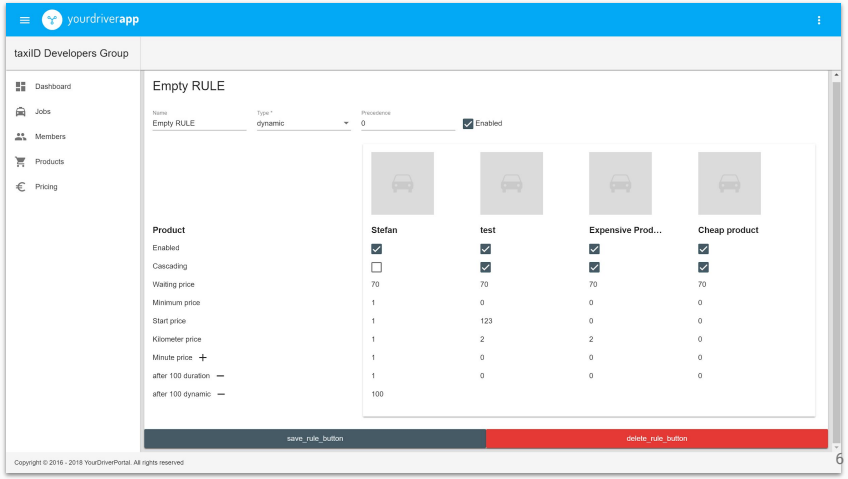


Pricing Detail Page

The detail page displays complex properties for each company product

Threshold prices have not been implemented fully, as can be seen in the lowest row of the table

When the table overflows, it becomes scrollable in the horizontal direction



Price Calculator Classes

Improvements have been made to the DynamicCalculator and FixedCalculator classes that are responsible for:

1. The reading of pricing information
2. Define which metrics are calculated
3. Contain defintion of a calculation
4. Generate high order calculator functions with default params
5. Call the calculate total function that calculates the total price using the generated calculators
6. Return the appropriate final amount

Dynamic Calculator

```
/**
 * Main calculation for a dynamic calculation.
 */
public static calculate(
  pricing: pricing,
  metrics: metrics & indexed,
): number {

  const thresholds: threshold[] = pricing.prices.dynamicThresholds;
  const startAmount = pricing.prices.dynamicStartPrice;
  const minAmount = pricing.prices.dynamicMinimumPrice;
  const cascaded = pricing.prices.cascadingThresholdCalculation;

  // Price for each metric
  const prices: indexed = {
    distance: pricing.prices.dynamicDistancePrice,
    duration: pricing.prices.dynamicMinutePrice,
  };

      ...
```

```
      ...
  // Explain how the price should be calculated
  const func = (price: number, unit: number) => price * unit;

  // Calculator for each metric using prices as default params
  const calculators = Calculator.generateCalculators(
    prices, metrics, func);

  // If thresholds have been provided, check if some metrics have
  // surpassed these thresholds per metric type, and calculate
  // the price using a provided function for highest or all
  // thresholds.
  const total = Price.total(
    metrics, thresholds, calculators, cascaded);

  // Return biggest of the two
  return Math.max(total + startAmount, minAmount);
}
```

Fixed Calculator

```
/**
 * Main calculation for a fixed calculation.
 */
public static calculate(
  pricing: pricing,
  metrics: metrics & indexed,
): number {

  const thresholds: threshold[] = pricing.prices.fixedThresholds;
  const cascaded = false;

  // Price for distance metric only
  const prices: indexed = {
    distance: pricing.prices.fixedPrice,
    duration: 0,
  }
      ...
```

```
...
// Explain how the price should be calculated
const func = (price: number, unit?: number) => price;

// Calculator for each metric using prices as default params
const calculators = Calculator.generateCalculators(
  prices, metrics, func);

// If thresholds have been provided, check if some metrics have
// surpassed these thresholds per metric type, and calculate
// the price using a provided function for highest or all
// thresholds.
const total = Price.total(
  metrics, thresholds, calculators, cascaded);

// Return biggest of the two
return Math.max(total, 0);
}
```


Meter Calculator

```
/**
 * Main calculation for a on meter calculation.
 */
public static calculate(
  pricing: pricing,
  metrics: metrics & indexed,
): number {
  return 0;
}
```


Data Management

1. When a product is created, pricings (pricing, dynamicPricing, fixedPricing) are automatically attached to each rule.
2. When a rule is created, pricings are created for each product.

Reflection - Must Haves

1. Thresholds that are incrementally bigger should be added to each pricing of a rule if the add button is pressed.
2. A threshold of a rule should be deleted for each pricing with the click of a button.
3. An error is sometimes shown when a particular combination of fields is mutated and saved.
4. Display prices in € instead of cents.
5. Test pricing calculation with data inserted by the user.

Reflection - Could Haves

1. Before leave warning, when a user has modified a form.
2. Order rules by precedence automatically.
3. Set precedence by dragging rules up and down.

B.6 Sprint 4 - Apps and Timeframes

Sprint 4
Apps and Timeframes

Apps, time limited pricing rules and special rates, on-meter price calculations

PM2 / TS Compatibility

Best case scenario: (on server)

```
1. $ npm install pm2@latest -g
2. $ pm2 update
3. $ pm2 install typescript@2.6.2
```

Worst case scenario solution: (locally)

1. Transpile .ts files locally
2. Push resulting .js files to the repository
3. Run project like normal on the server

Timeframes Proposal

Timeframes were added as a definition of time restrictions to rules and discounts by disabling and enabling hours of a repeating week schedule, between two timestamps

The hours of the week as binary

1. The hours that are active within a week can be stored as bits.
2. Each day would have 24 bits, so a week would have 24 * 7 = 168 bits.
3. 168 / 8 = 21 bytes of data storage would be required.
4. Scalability is limited by the ability to bulk update existing binary values, for example: when in the future half hours should be represented instead of hours. Updating from 21 bytes to 42 bytes.

Test 1: BinData

1. See the BSON spec for more information about BinData(number, string)
2. Type 0 would allow an arbitrary length of data to be stored.
3. Loopback stores encoded strings, making it hard to store and retrieve data.
4. When forcing loopback to store as buffer, the data cannot be read, and must manually be modified.

Inserting

Type 0 BinData is a generic BinData type that accepts generic formats

```
db.Timeframe.insert({

   startDate: new Date(2018, 4, 7),
   endDate: new Date(2019, 4, 7),
   weekSchedule: BinData(0,

     "00110100011001101100001 1
      01101011001100001011110 0
      10101010111010001111100 0
      11111001111101110010000 1
      10100000001011101110010 0
      11001000000100001010110 1
      01011101000000101001110"

   )
})
```


Test 2: boolean[]

1. A boolean[] can be queried by indexes, which makes it easy to find results where the indexes have a truthy value.
2. The query however must be constructed dynamically by adding clauses for each index. Errors may be thrown when indexes don't match.
3. This solutions makes it hard to check more than one index, having to build complex queries in advance.

Inserting

An array of booleans can be quite verbose

```
db.Timeframe.insert({

   startDate: new Date(2018, 4, 7),
   endDate: new Date(2019, 4, 7),
   weekSchedule: [
      true,
      false,
      true,
      ...      // 162 more
      false,
      false,
      false
   ]

})
```

Querying

The resulting query matching values verbosity increases depending on amount of indexes checked

```
db.Discount.find({

  timeframes: {

    weekSchedule.3: true,
    weekSchedule.16: true,
    ...
    weekSchedule.128: true,
    weekSchedule.129: true,

  }

})
```


Test 3: string

1. A string is a very flexible datatype.
2. Using a regex in a query makes checking multiple bits in the string relatively easy, and enables different values next to 0 and 1.
3. It also makes querying the data really stable, as the query will silently fail if the content of the data is not of expected length or value.
4. Performance is not an issue if the regex column is indexed, and when prefix expressions (/^/) are used: docs.mongodb.com/manual/reference/operator/query/regex/#index-use
5. Another advantage is freedom and scalability. If multiple values, or ranges need to be matched, a simple regex modification is sufficient.

Inserting

1. startDate and endDate: absolute boundaries of the schedule
2. weekSchedule: every 24 bits represent 24 hours of a day of the week

```
db.Timeframe.insert({

   startDate: new Date(2018, 4, 7),
   endDate: new Date(2019, 4, 7),

   weekSchedule:

    "001101000110011011000011  //m
     011010110011000010111100  //t
     101010101110100011111000  //w
     111110011111011100100001  //t
     101000000010111011100100  //f
     110010000001000010101101  //s
     010111101000000101001110" //s

})
```


Querying

Patterns matching fields support indexing, improving performance

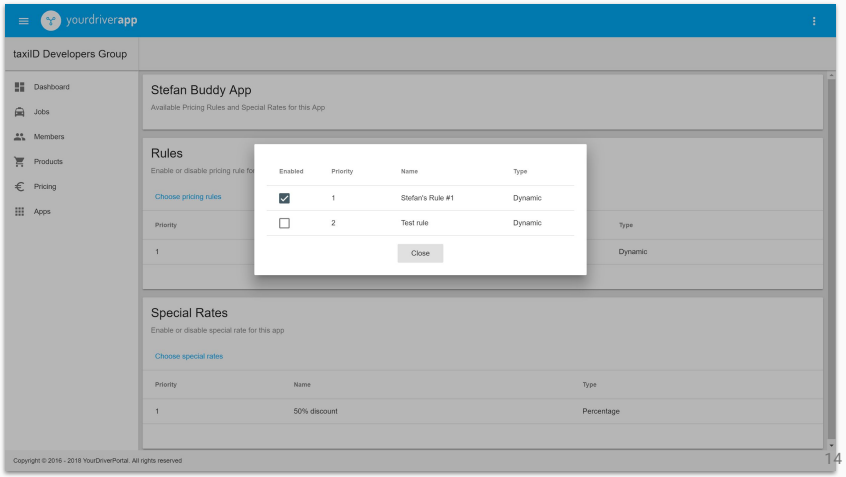
```
// First bit is 1
Db.Discount.find({

   "timeframes.weekSchedule": {
     $regex: /^1/
   }

})

// 2 4 6 8 10 12 15 bits are 1
/^.1.1.1.1.1.1..1/

// 128 129 and 131 bits are 1
/^.{127}11.1/
```

Views

The following slides show the views that were added during this sprint

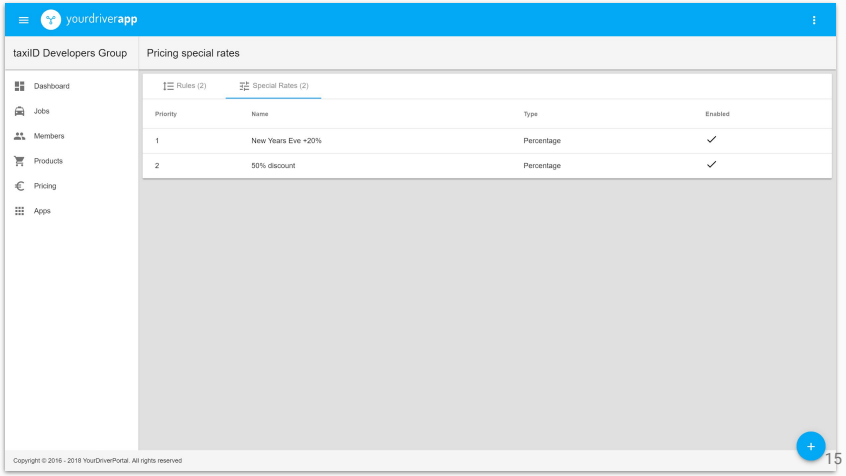
Apps, Enabled Rules and Special Rates

A modal opens up when ‘Change x’ is clicked, showing a table that allows users to add or remove pricing rules and / or special rates to the App



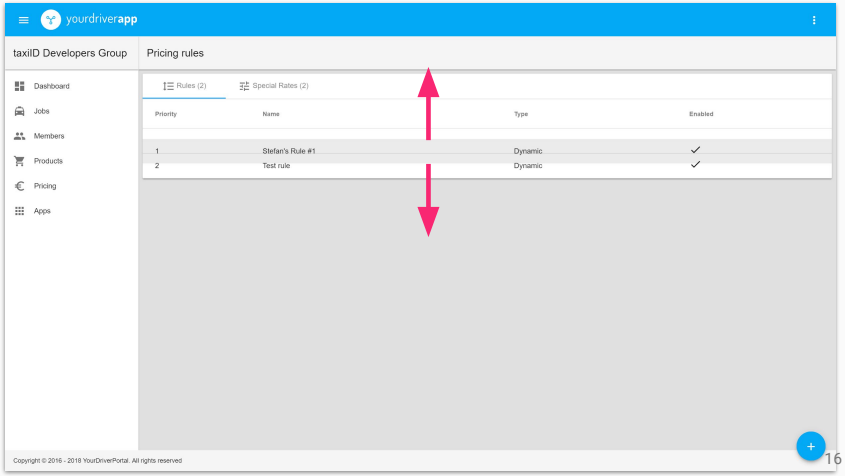
Pricing Rules / Special Rates

The Pricing tab now shows two tabs. Both having a table in which the rows can be sorted, using the exact same logic



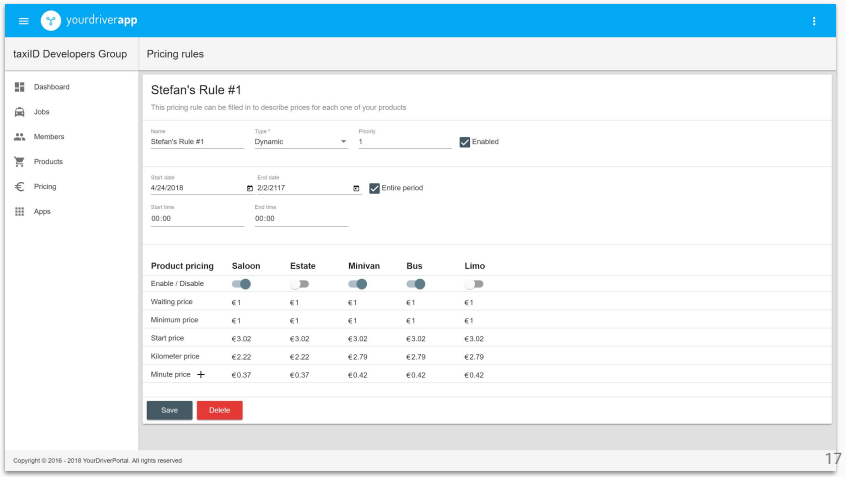
Dragging Pricing Rules / Special Rates

Automatically sorts stuff in the backend when an entitiy is picked up and dropped at some location



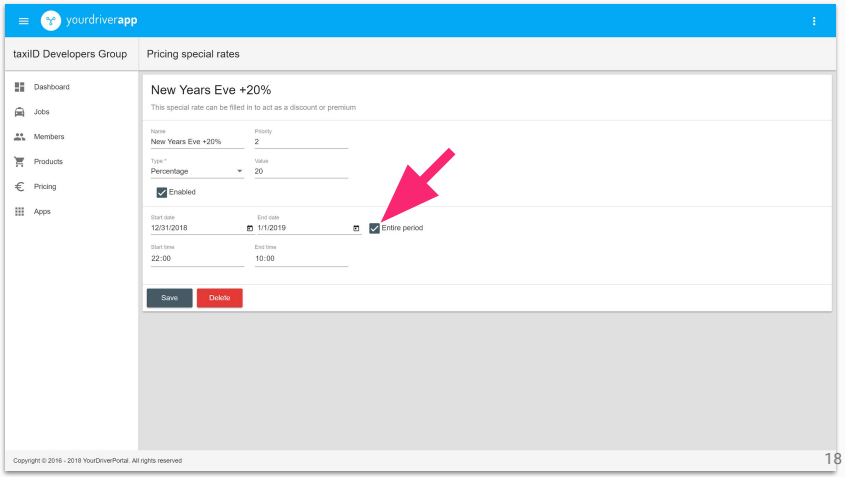
Pricing Rule

A timeframe is added to the pricing rule as a component



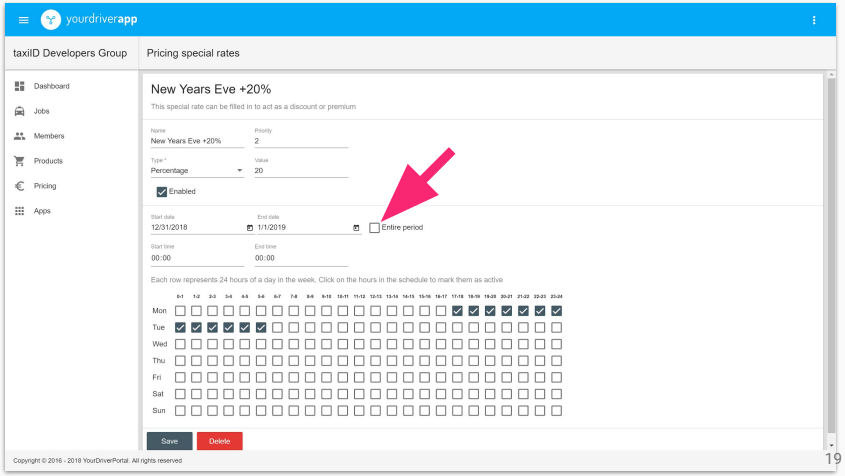
Special Rate

The Special Rate page is added, having the same timeframe component as the Pricing Rule page



Special Rate

Alternatively, when the ‘Entire Period’ checkbox is unchecked, the hour selector for the week is shown, allowing the user to customize every single hour of the week, from start till end date



Data Management

1. When a new pricing rule or discount is created, its priority will be 1, other element priorities get incremented
2. When an element is deleted, all elements with larger priority get decremented
3. When an element is modified, all other elements are modified so that the priority order *1* … *n* is maintained
4. A priority that is given, higher than *n* is capped at *n*
5. A priority lower than *1* is defaulted to *1*
6. A timeframe is added by default

Reflection - Must Haves

1. Thresholds that are incrementally bigger should be added to each pricing of a rule if the add button is pressed
2. Thresholds should be deletable with a simple button click
3. Display fixed special ratings and fixed threshold prices in € instead of cents, display percentages with the % symbol
4. Pricing Rules are stored in the database before save is pressed for technical simplicity. This should not be the case ideally

Reflection - Could Haves

1. Show warning if user has modified form fields, and tries to leave the page
2. Remove isEnabled flag for pricing rules and special rates, as they have to be enabled in the Apps view as well

B.7 Sprint 5 - Thresholds

Sprint 5
Thresholds

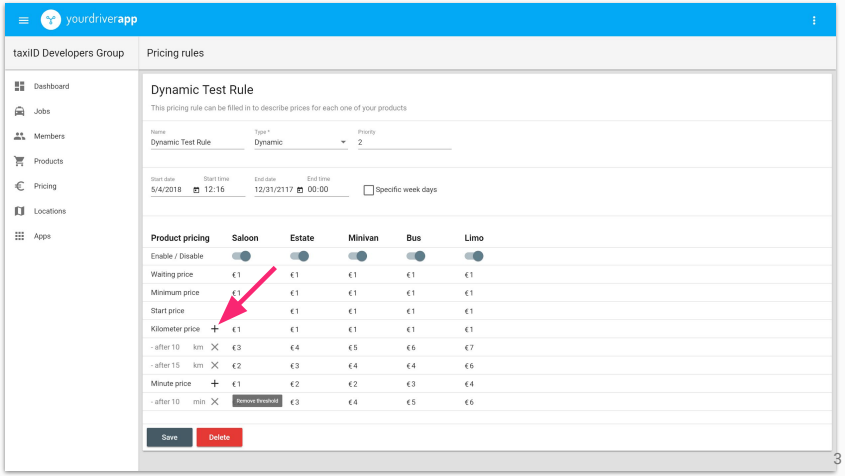
Improving thresholds, specific week days, price estimations, portal authentication, and implementing location to location rules

Thresholds

1. Thresholds are embedded as arrays in the fixedPrice and dynamicPrice entities
2. Thresholds do not rely on id's
3. Thresholds are sorted before they are added based on threshold property ASC
4. There must not be duplicate thresholds, ex: that there are two thresholds at 15 min
5. Thresholds can be edited, but will not automatically sort in the frontend

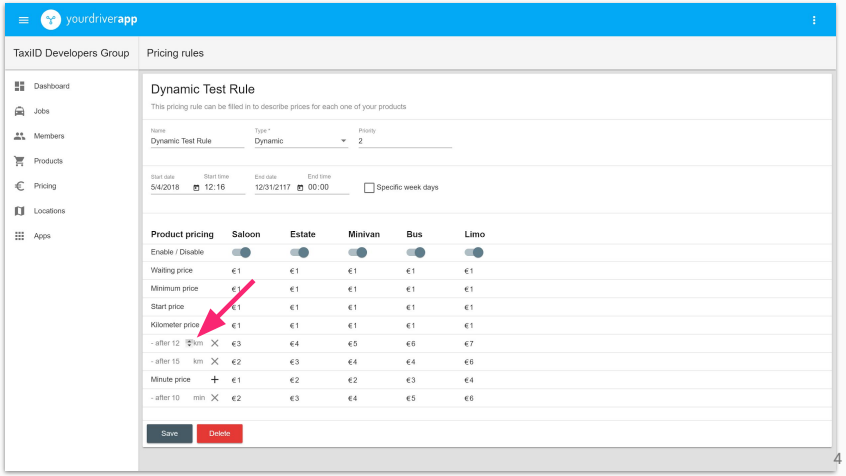
Thresholds

The + icon adds a threshold under the kilometer or minute price. The x icon removes a threshold.



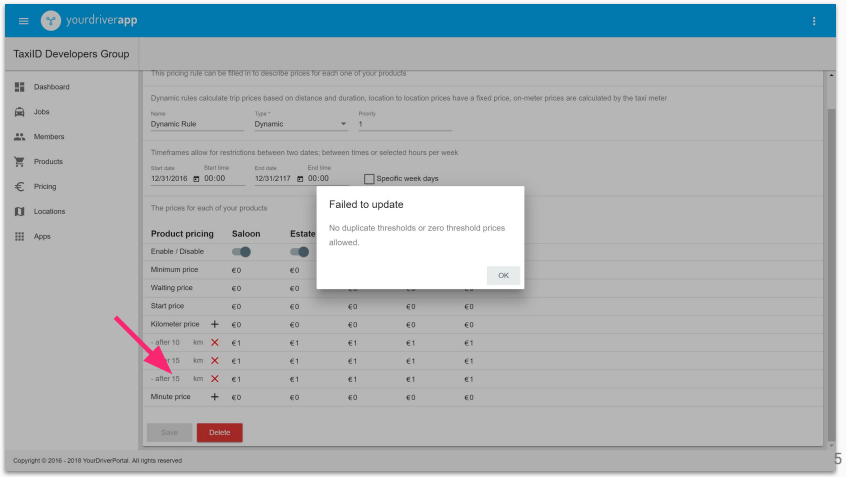
Thresholds

The threshold property can be modified, just like the prices in each product column.



Thresholds

A warning is shown if thresholds thresholds exists, or if zero threshold prices are submitted.



Thresholds, functional filter

```
/**
 * Check all prices for no duplicate thresholds etc.
 */
getDuplicates = (array: any[]) => array.reduce((acc, el, i, arr) => {
  if (arr.indexOf(el) !== i && acc.indexOf(el) < 0) { acc.push(el); }
  return acc;
}, []);
noDuplicateThresholds = (prices: Price[]) => prices.every(p =>
  ['priceDynamic', 'priceFixed'].every(priceType =>
    ['distance', 'duration'].every(metricType =>
      !this.getDuplicates(p[priceType].thresholds
        .filter(t => t.type === metricType)
        .map(t => t.threshold)).length)));
noZeroThresholdPrices = (prices: Price[]) => prices.every(p =>
  p.priceDynamic.thresholds.every(t => t.value > 0)
  && p.priceFixed.thresholds.every(t => t.value > 0));
```

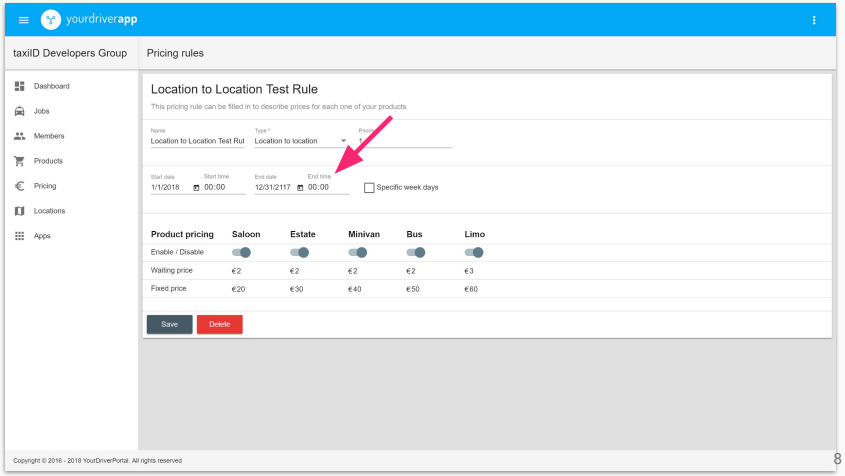
```
if (
  !this.checkIf(this.form.value.prices,
    this.noDuplicateThresholds,
    this.noZeroThresholdPrices)
) {
  this._dialogService.openAlert({
    title: 'Failed to update',
    message: 'No duplicate thresholds or ...,
    disableClose: true,
    closeButton: 'OK',
  });
}
```


Specific Week Days

1. When specific week days is unchecked, start time and end time can be specified
2. When specific week days is checked, the usual timeframe selector is displayed

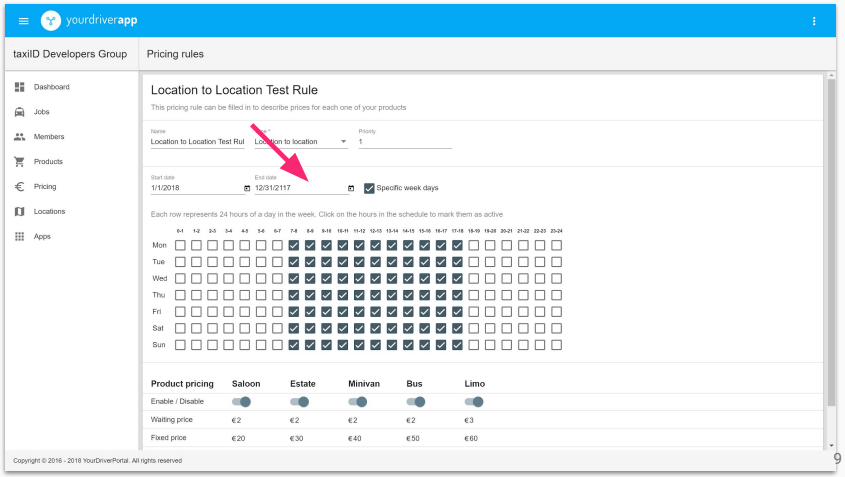
Specific Week Days

When the specific week days option is disabled, start and end time can be defined to make the date definitions more precise



Specific Week Days

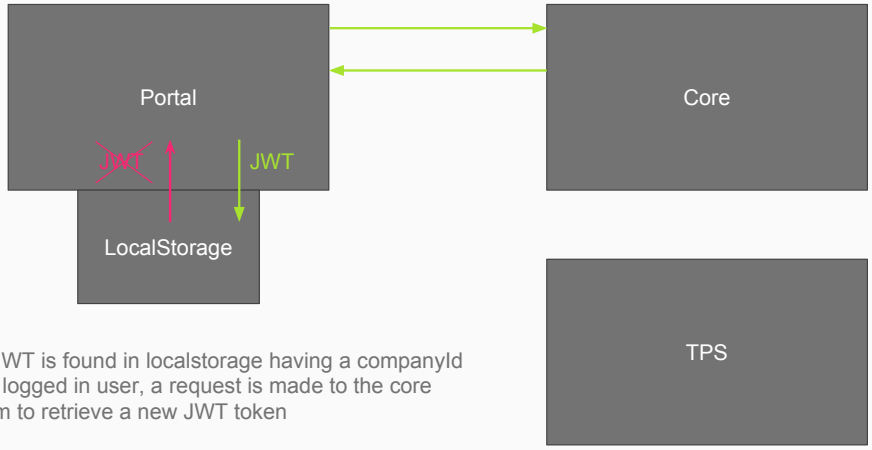
When the specific week days option is enabled, days in the week can be defined more precisely with the granularity of an hour



Price Estimations

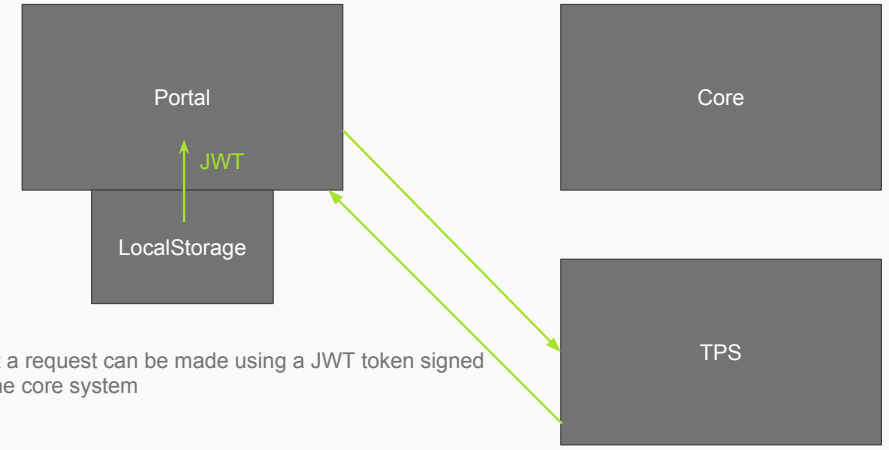
1. Relations between the following entities have been made polymorphic hasMany through:
   a. DaAppInstall - Rule
   b. DaAppInstall - Discount
   c. Debtor - Rule
   d. Debtor - Discount
2. The relations are defined as models called links: ruleLink & discountLink
3. The link between rule and DaAppInstall has a property 'isEstimated': true | false

Portal Authentication



If no JWT is found in localstorage having a companyId of the logged in user, a request is made to the core system to retrieve a new JWT token

Portal Authentication



Next a request can be made using a JWT token signed by the core system

Automatic Company and DaAppInstall Sync

1. An endpoint allows companies and daAppInstall entities to be created in TPS
2. The following properties are required:
   a. companyId
   b. daAppInstallId
   c. name
   d. country: ISO Alpha-2 country code
3. The Core tries to keep TPS up to date on companies and daAppInstalls that are added

B.8 Sprint 6 - Locations

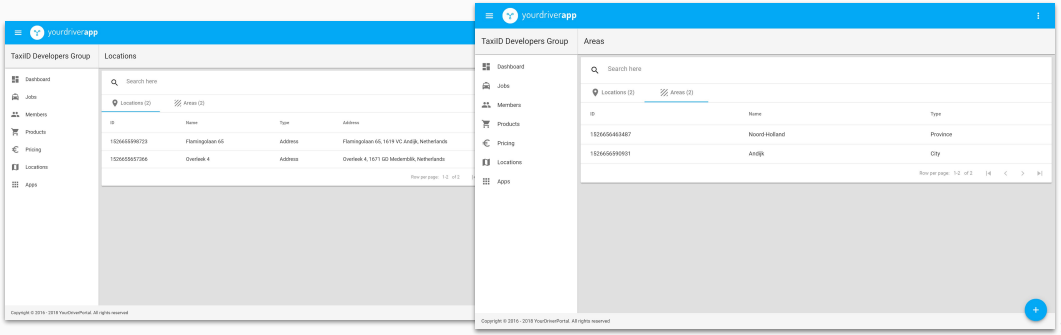
Sprint 6
Locations

Searching, importing, and drawing polygons to define locations

Location Overviews

Locations

Areas



Location Detail Views

Locations:

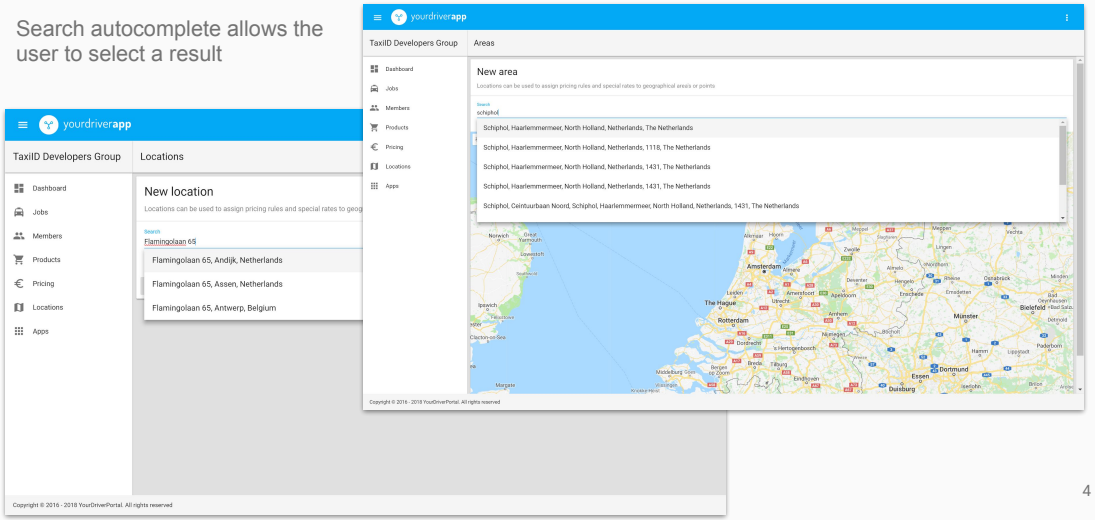
1. **Search** location from https://www.google.com/maps
2. **Select** location from the search results
3. **Import** small polygon around the point
4. **Edit** the shape, position, name and type
5. **Save** location

Areas:

1. **Search** location from https://nominatim.openstreetmap.org/
2. **Select** location from the search results
3. **Import** polygon
4. **Edit** the shape, position, name and type
5. **Save** location

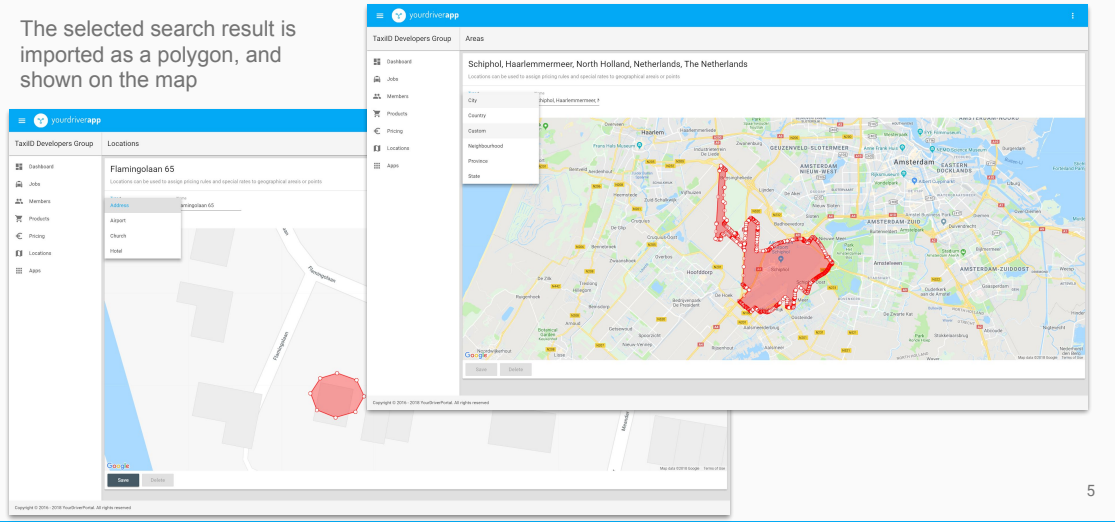
Search

Search autocomplete allows the user to select a result



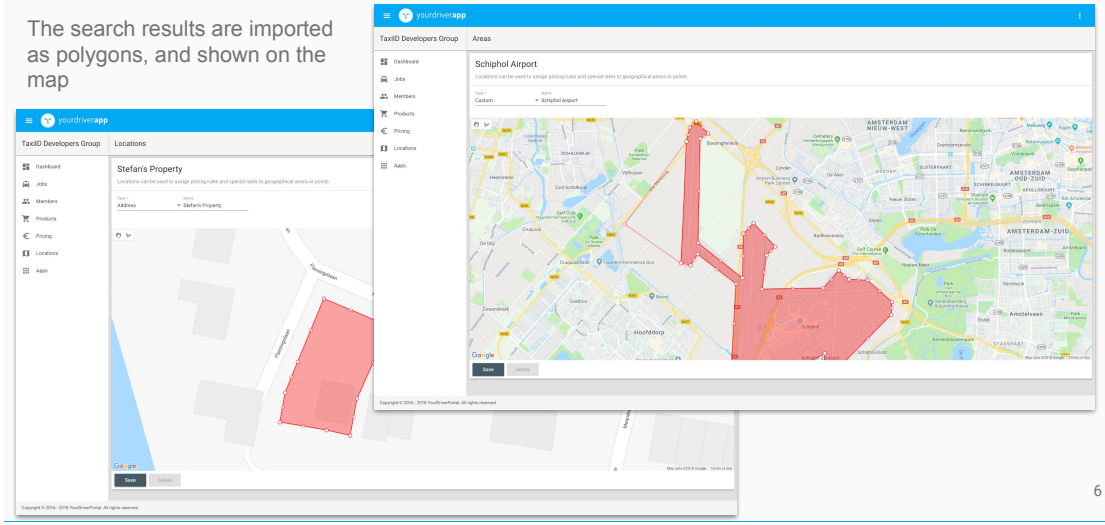
Select and Import

The selected search result is imported as a polygon, and shown on the map



Edit and Save

The search results are imported as polygons, and shown on the map



Location Searching Filters

Google locations can be filtered on the following types:

1. Geocode
2. Address
3. Establishment
4. Regions
5. Cities

https://developers.google.com/places/web-service/autocomplete

The OpenStreetMaps counterpart does not allow filtering yet:

https://wiki.openstreetmap.org/wiki/Nominatim

Location Searching Restrictions

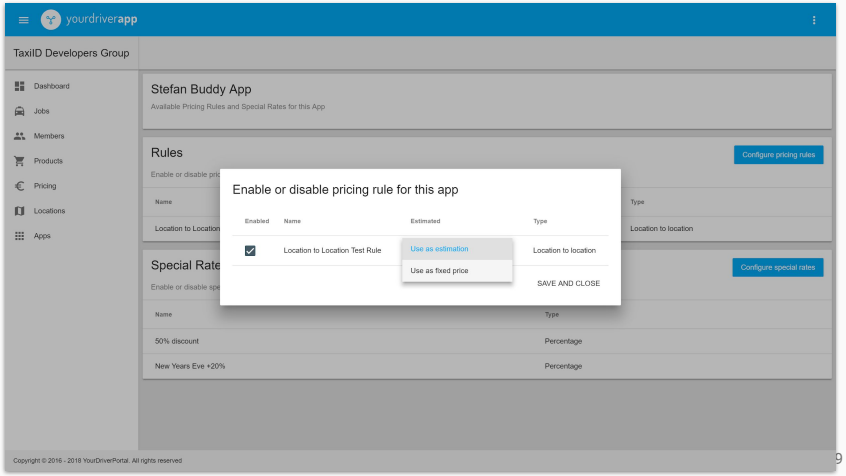
https://operations.osmfoundation.org/policies/nominatim/ forbids auto-complete search, but bulk queries are allowed while limited to one thread, one machine, and cached

For this reason, it is advised to use Google Places for searching. When a place has been found, the polygon data should be imported once

Alternatively, datasets could be downloaded at https://planet.openstreetmap.org/ and used locally

Estimations Select

A select dropdown communicates the intentions of the setting more directly over a checkbox



B.9 Sprint 7 - Subrules

Sprint 7
Subrules

Improving usability of entering large amounts of fixed rules

Dynamic Rule Locations

A dynamic rule may optionally have a Departure and Destination location

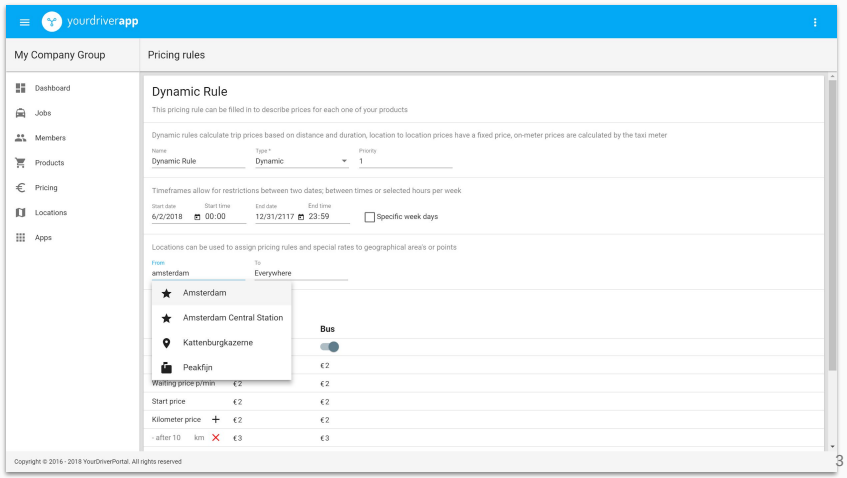
If a location is left empty, the location is assumed to be ‘Everywhere’

When the user starts typing in the location field, suggestions of already created locations are displayed, including locations created by developers, which are available to all users

If no suggestion is found, but the user enters the value, a propt asks the user to create a new location

Dynamic Rule Locations

Search suggestions are based on the location name and address



Fixed Rule Locations

A fixed / location to location rule can have multiple subrules to speed up creation of each price definition, having:

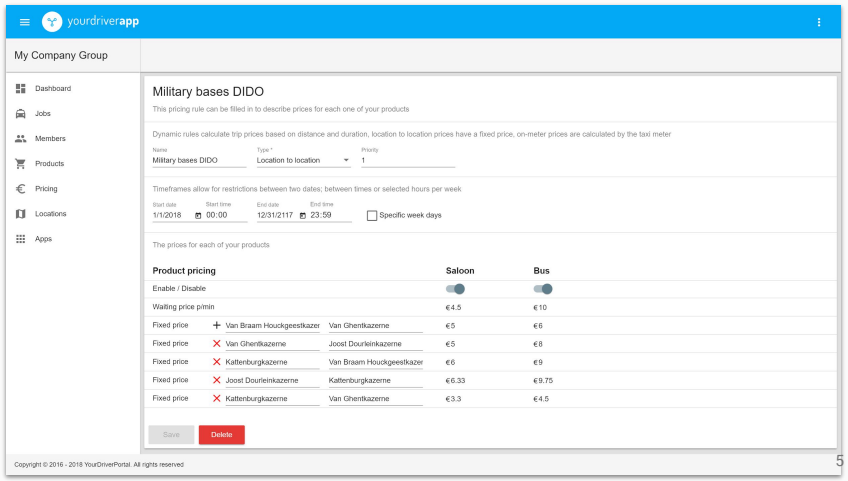
1. Departure
2. Destination
3. Fixed price for each product

Departure and Destination are required properties

Fixed Rule Locations

Like the threshold rows in the dynamic price rules, sub rules can be added by clicking the plus sign

A fixed rule must have both departure and destination locations defined



On-meter

A change to estimations and on-meter prices is bringing the settings to the Apps view

If a location is not provided by the bookings app, no price can be calculated, and therefore users must be able to define whether they would like prices to be calculated in a later stage

The table in the next slide shows eight situations, whether products are shown, whether prices are calculated or estimated, and if meter prices can be used

= on = off = impossible

show - products are shown, regardless of whether prices have been calculated for them

prices - prices have been calculated

estimation - prices have not been calculated

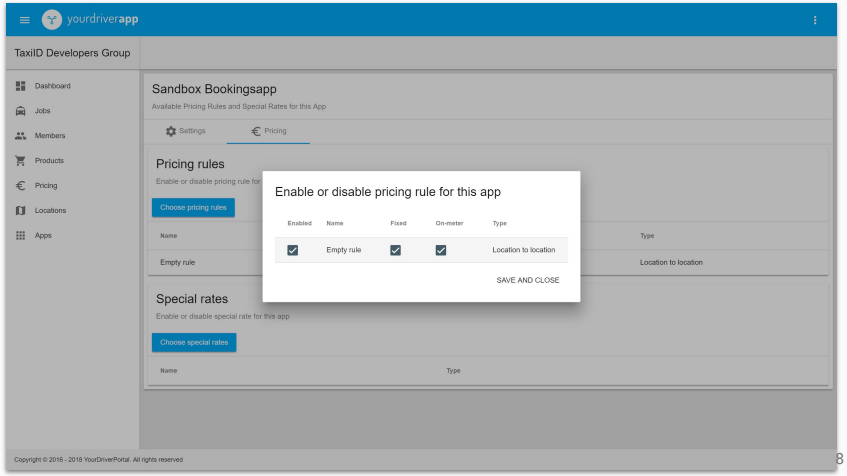
on-meter - allow app to determine the price on meter

Both locations provided	Is fixed	Is on-meter
show →	prices	on-meter
show →	prices	
show →	estimation →	on-meter
show		
show		

Eight situations depending on settings and whether a destination is provided

Estimations Select

If a rule is enabled, the fixed and/or the on-meter checkbox must be enabled



Authorization

A token is requested from the core API whenever group routes are being accessed

This token is stored in localstorage

The token is sent to TPS on every request

TPS limits the user's access to resources that are not bound to the JWT payload identity

Authorization - Requesting JWT

```
/**
 * See if route can be activated.
 */
canActivate(
  route: ActivatedRouteSnapshot
): Observable<boolean> | Promise<boolean> | boolean {

  // Identity information from url and local storage
  const companyId = route.params.id;
  const daAppInstallId = this.getFromStorage(
    vaultPrefix + '.driver',
    'daAppInstallId'
  );

  ...
```

```
  ...
  // See if JWT exists, or request a new one
  const jwtName = jwtKeyName(companyId);
  if (this.itemIsSet(jwtName)) return true;

  // Request new JWT and return the status as a boolean
  return this._companyService.getNewJWT(companyId, daAppInstallId)
    .map(response => {
      this._vault.setItem(jwtName, 'Bearer ' + response.jwt);
      return this.itemIsSet(jwtName);
    })
    .catch(() => Observable.of(false));
}
```


Authorization - Sending JWT in Headers

```
/**
 * Retrieve JWT token as headers.
 */
getJWTHeaders = (): HttpHeaders => {

  const companyId =
        this._router.routerState.snapshot.root.firstChild.params['id'];

  const bearer = this._vault.getItem([
    environment.vaultPrefix,
    companyId,
    'jwt',
  ].join('.') || 'Bearer ');

  return new HttpHeaders()
    .set('Accept', 'application/json')
    .set('Authorization', bearer);
}
```


Authorization - Authorizing Request

The JWT payload id's are added to the query filters to only allow certain resources to be accessible

```
// Ensure filter exists
if (!ctx.args.filter) ctx.args.filter = {};
if (!ctx.args.filter.where) ctx.args.filter.where = {};

// If belongsTo company, force JWT token companyId
if (relatedToCompany) {
  if (ctx.args.data) ctx.args.data.companyId = companyId;
  ctx.args.filter.where.companyId = companyId;
}

// If belongsTo daAppInstall, force JWT token daAppInstallId
if (relatedToDaAppInstall) {
  if (ctx.args.data) ctx.args.data.daAppInstallId = daAppInstallId;
  ctx.args.filter.where.daAppInstallId = daAppInstallId;
}
```

B.10 Sprint 8 - Polishing

Sprint 8
Polishing

Improving performance, usability, maintainability and extensibility

Final Feedback

1. A user must be able to copy locations from a list of predefined location, so that they can be used or customized for a particular company
2. The option to enable rules in the Apps detail page can be removed, because two checkboxes already cover the possibility of disabling a rule
3. The breakdown results should be ordered by vehicle type