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

May 2018

Todo list

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*

Abstract

A purely geometrical interpretation of user-defined locations would allow taxi-companies around the world to set up rules so that trip prices could be calculated without depending on distinct postal code systems. Geolocation datatypes provide part of the solution, but the 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

Chapter 1

Introduction

What was once an ordinary startup known as Uber, is now the most famous taxi dispatch company in the world [1]. In the same year that Uber was founded, a similar startup in the Netherlands called taxiID was launched; an Amsterdam based company providing end-to-end cloud solutions 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 everyones 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 matter 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 intoduces complicated challenges when developing applications that rely on clearly defined locations and infrastructures, often vastly differing between countries, if these countries have such a system to begin with. The taxiID development team responsible for solving these problems is located in Medemblik. Consisting of two mobile app developers (iOS and Android), two backend developers, a designer and two project managers.

1.2 Problem Definition

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

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

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

in chapter 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, using a newly referenced fixed pricing rule. This process is not just hard to understand for a user, who has to reason about the companies prices. But it is also hard to understand for programmers, who have to maintain the code that supports this functionality. A small mistake in the csv file could lead to great issues if the mistake goes through processing undetected.

1.3 Assignment

The title of this thesis reads:

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

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

1.4 Research

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

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

1.4.1 Questions

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

How can a generic location-based price calculation system be implemented

that could be used in every country?

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

1. In what way can locations be represented to be universally interpretable and precise?
    1.1. Which location types matter for this project?
    1.2. How can postal codes be abstracted to geospatial data while retaining the same usefulness in the system?
    1.3. Can a system be created that does not rely on postal codes and addresses?
    1.4. Which Database Management Systems (DBMS)s cover the location storage use cases for this project?

2. What is most fitting solution to integrate the backend and frontend into the existing architecture?
    2.1. Which architectural patterns fit in with the exising architecture?
    2.2. Which data of adjacent systems are required to make TPS operational?
    2.3. How can authentication between services be implemented or improved?
    2.4. Which methods and technologies can be used to ensure suitability and improve maintainability and testability?
3. Which logic and information is required to calculate a trip price breakdown?
    3.1. How are rules matched?
    3.2. Which parts of the system have an impact on the calculation result?
    3.3. How should the concept of time schedules be realised?
4. How can complex pricing rules be communicated through the UI?
    4.1. Which views are essential?
    4.2. In what way can visual hierarchy guide the user through processes naturally?
    4.3. How should complex elements impacting price calculations be communicated to the user through UI components?

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. This document clearly documents the interpretation of the product owners product vision and reflects from a developer viewpoint, 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

Encoding of locations has historically been of great importance, and is always being modernized. This chapter explains the general definition of locations, which types of locations are important for this project, and how to represent these locations so that they are universally interpretable and do not rely on a postal code system.

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 marinner'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. This particular system is still commonly used today.

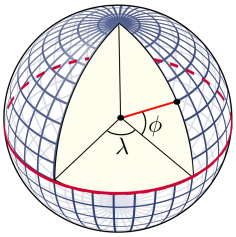


Fig. 2.1 A perspective view of the Earth showing how latitude and longitude are defined on a spherical model.

The history of this encoding goes way back to when it was first proposed in the 3rd century BC by Eratosthenes. He invented the discipline of geography, and was known for also being the first person to calculate the circumference of the Earth with remarkable accuracy. Today, navigation relies on satellites that are capable of providing information to determine a location with a precision of 9 meters. Hybrid methods using cell towers, Wi-Fi Location Services, and the new Galileo global navigation sattelite 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 radiusses and shapes. You could prepend 'the location of' to the following terms as an example: America, the birthplace of Sokrates, Wall Street, the center of the universe, the Laryngeal Nerve of the Giraffe, churches in the Netherlands. The final example presents the main challenge of this project, how to communicate the location of a collection with points or areas of differing shapes and sizes that may overlap?

2.3 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 infitesimal. Only when decimals were disregarded to decrease the precision of a point it would be possible, in which case it would still not be usefull in this application, because the imprecise point would be a square.

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 as it would 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 sould 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. 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 a subset of points, called an area, is in all possible areas A:

$$a_1 \in A$$

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. 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.

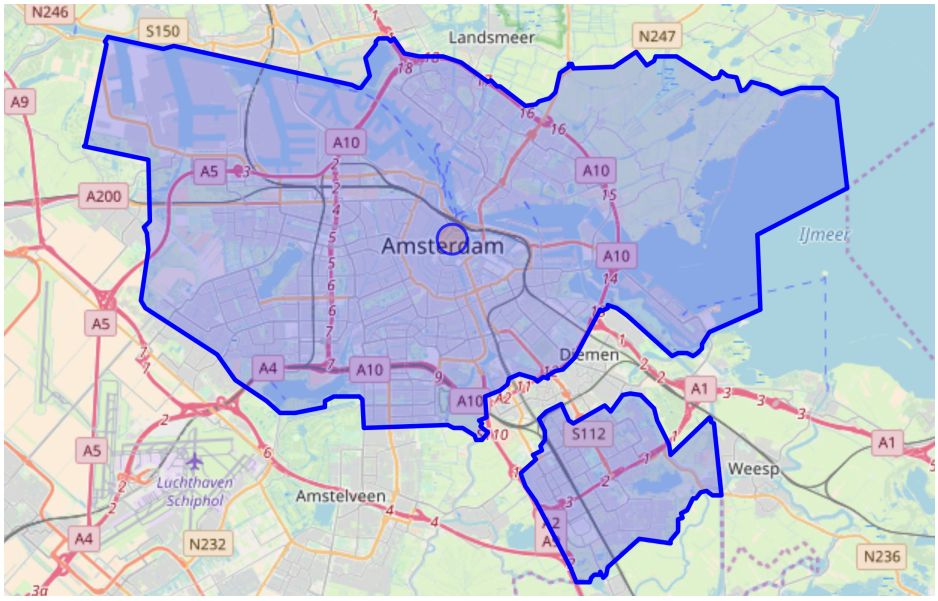


Fig. 2.2 Amsterdam - A single location comprised of multiple locations.

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.

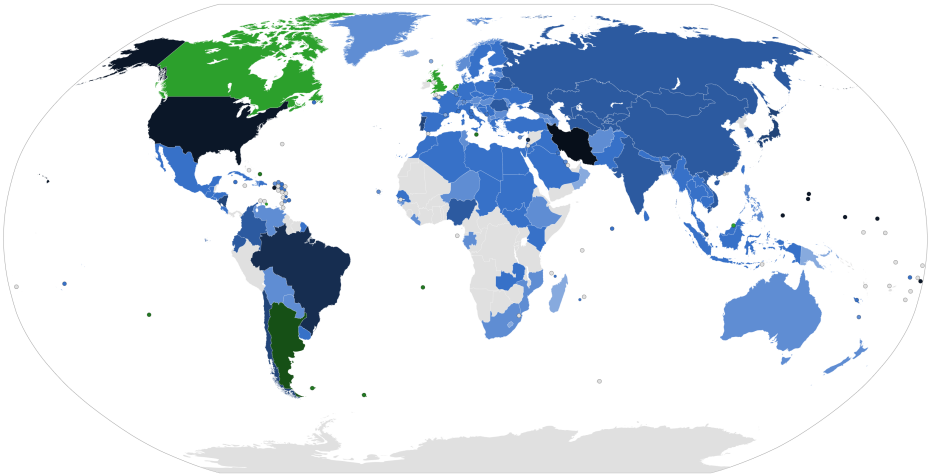


Fig. 2.3 Postal codes by country with amount of digits ranging from three digits (lightblue) to eight digits (darkgreen) and no postal code system (gray).

When a trip is booked in the United Arab Emirates, the only information provided as a destination or departure location are geospatial coordinates and addresses. These addresses are compared with long lists of matching addresses in the legacy database, as previously described with postal codes. 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. Addresses can be ambiguous, addresses and postal codes can be imprecise, postal codes are not uniform, and some countries do not have a postal code system. In contrast, polygons would provide unique and precise location definition that is uniform and universal. But postal codes and addresses have been useful in the legacy system. 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 deterministic predicate of whether a location is fully contained within a location is achievable
4.	A deterministic 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

Many spatial database systems support a basic Geometry hierarchy of Points, Polygons, MultiPoint and MultiPolygon Classes, as described in the OGC [3] and ISO 19125 [4] standard. These spatial datatypes would could be stored as single entities. 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. Databases like MySQL and PostgreSQL provide similar functionalities that adhere to the OGC standard, while MongoDB has its own implementation. These functionalities can be seen as a straight forward approach in solving geometry based queries. What 3 words, a multi-award winning global addressing system, bases 3m x 3m squares, covering the planet, on a combination of three words. In their whitepaper: "Efficient and future-proof", CEO Chris Cheldrick explains how locations can be communicated more effectively by describing a three by three meter areas using three words [5]. The what3words API offers functionalities that can find what3word geocodings near a specified latitude and longitude location. It is also able to find results within a clamped area, as documented in [6]. This system could be used to retrieve area's as three words in a database with given clamp parameters, which could then be stored in the database as single entities. This system would solve the problem of having ambiguity in address or postal code systems, and it is also very accurate. But in order to provide complex polygon matching, a spatial database system is still required. If a radius was assigned to the centroid of the shape that is formed by the boundaries of the street, neighbourhood, province or country, the encoding could technically be defined as a what3words query. A combination of clustering

of points around what3word encodings could provide a system that is far less accurate, but is usable.

kks32: Keep trying to find if other people found solutions for determining precedence of polygon matches

http://geoawesomeness.com/discrete-global-grid-system-dggs-new-reference-system/

2.5 Database Prerequisites

The database that is used must be able to determine whether a polygon contains a given point, and must be able to aggregate all points that are contained within a given polygon. The scenario presented in image 2.4 should be replicable.

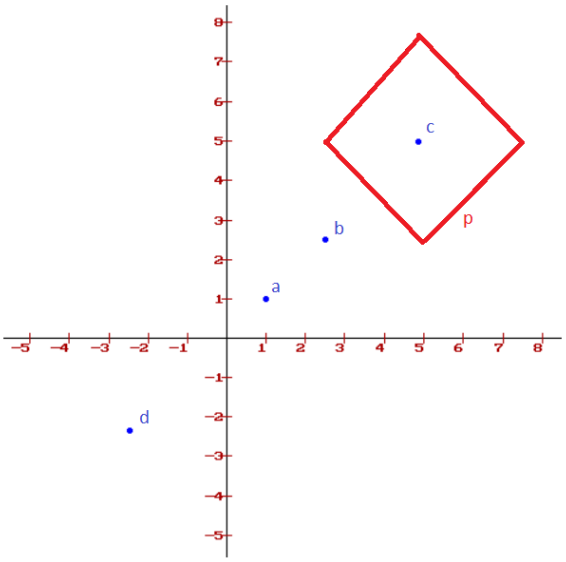


Fig. 2.4 Four Points, one Polygon p containing Point c.

As contained in Appendix A, this example provides a proof that a minimal requirement is satisfied, so that a list of candidate Database Management Systems could be constructed. 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 [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 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.

```
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 [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 2.4 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 an extra pair of braces, as shown in 2.6, while retaining the 'polygon' shape type.

```
db.polygon.insert({
  shape: {
    type: "Polygon",
    coordinates: [ [ [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

Priority precedence polygon size arrival vs departure clustering around hotspots

kks32: Currently implementing solution

2.7 Performance and Clustering Trade-offs

Agarwal and Rajan state that NoSQL take advantage of cheap memory and processing power, thereby handling the four V's of big data more effectively, but lack the robustness over SQL databases in [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, as per [11]. After the 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.

2.8 Conclusion

Every point and area on earth can be described by geometric datatypes such as Polygons or MultiPolygons, yielding a consistent and universal solution of storing spatial data. This means that a new system does not have to rely on address and postal code data required by the legacy system. Some alternative encodings like what3words may be used to encode distinct points, but area's become quite a challenge in such systems. All physical locations located on the earths surface that were described by addresses and postal codes can be encoded by spatial datatypes. The multipolygon may be used to represent more complex areas like the city Amsterdam as seen in 2.2. The OGC standard offers a mature and thorough set of geospatial features that MongoDB does not support. If performance and expandability are important, a MongoDB NoSQL database will result in the better choice. Otherwise, MySQL, MariaDB, PostgreSQL and many other database systems offer GIS features that adhere to this projects requirements. Overlapping areas in conflicting rules can be resolved by assigning priorities to rules. This enables users to reason about their pricing rules more easily, and makes the intent of these pricing rules more interpretable for other people.

Chapter 3

System Architecture

3.1 Introduction

In order to succesfully integrate a new system component in an existing system architecture, flows of information must be aligned with adjacent system components, so that data dependencies of the new component are satisfied, and expected functionalities can be provided in return. This chapter explains which important aspects dictate the systems architecture to implement TPS, taking quality metrics into account. ISO/IEC 25010 is the latest objective standard for systems and software engineering, providing a common understanding of software quality characteristics [13].

3.2 Architectural Patterns

The current system architecture consists of three API's and nine services that connect to four databases, as can be seen in Figure 3.1. They provide functionalities to portals and mobile apps. The user interface, business logic, and data storage are separated, following the three-tier or multi-tier architecture as described in [14].

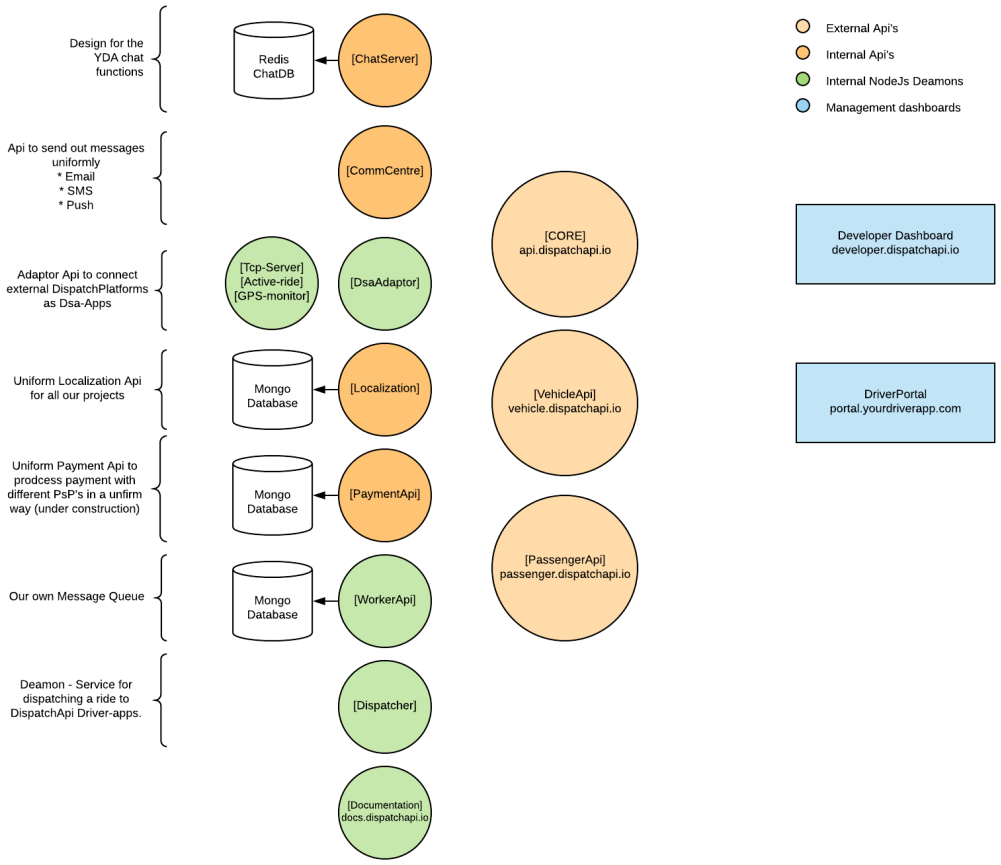


Fig. 3.1 Current System Architecture provided by taxiID

The bigger and smaller shapes in the Figure represent large API's and smaller services respectively. The orange colored services are used internally, the green ones can be used by other companies. The smaller services adhere to the pattern that is called service-oriented architecture (SOA), where application components provide services over a network typically.

3.2.1 Monoliths

Monoliths are large single upright blocks of stone, especially shaped into or serving as a pillar or monument — almost describing a single tiered software application. 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 [15]. In general, a monolith describes a software application which is designed without modularity. The bigger shapes in Figure 3.1 can be classified

as monoliths. 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. 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 in this manner, now facing difficulties of providing the price calculation functionality to new projects. If the previous price calculation system was implemented as a service, it could have been reused or redeployed as a second separate price calculation system for YDA.

3.2.2 Microservices

A consensual definition of microservices does not exist, but can be described 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 [16], mentioning; improvements in development speed through parallel development, isolated deployment and continuous delivery (CD), scalability and potential parallelism, and independence in case of failure. Fiar points of criticism have also been made in regard to microservices. Jan Stenberg has pointed out that microservices are information barriers in [17], 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

The requirements state that the frontend should be integrated in multiple portals. This would mean that separate views have to be developed for each portal, or the views should be provided to the portals via iframes, or some similar technique. In the last case, it may be benificial to combine the frontend and backend in the same project structure. 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. Possibilities of separation and integration of the frontend and the backend are another aspect that has to be taken into account before implementation of TPS.

3.3 Information Dependencies

TPS will provide two types of services based around the same data. Portal users can mutate pricing information, mobile apps can retrieve trip prices. To effectively calculate the price of a ride, or to allow the portal user to mutate pricing data, the app or portal sending the request must be identified and authorized, which will be discussed in the next section. Assuming this has succesfully been achieved, some data may or may not be required from other services or databases in the system architecture. In the case of a price calculation, some extra required data are sent in the body of the request:

1. vehicleTypes: string[]
2. passengerCount: number
3. requestedDate: ISODate
4. departure: { gps: { lat: string, lng: string } }
5. destination: { gps: { lat: string, lng: string } }

In both the price calculation and the portal data mutation case, required data are stored in one or more databases. The proposed database schema for TPS in Figure 3.2 shows the general structure of the data that must be stored for TPS in order to be operational.

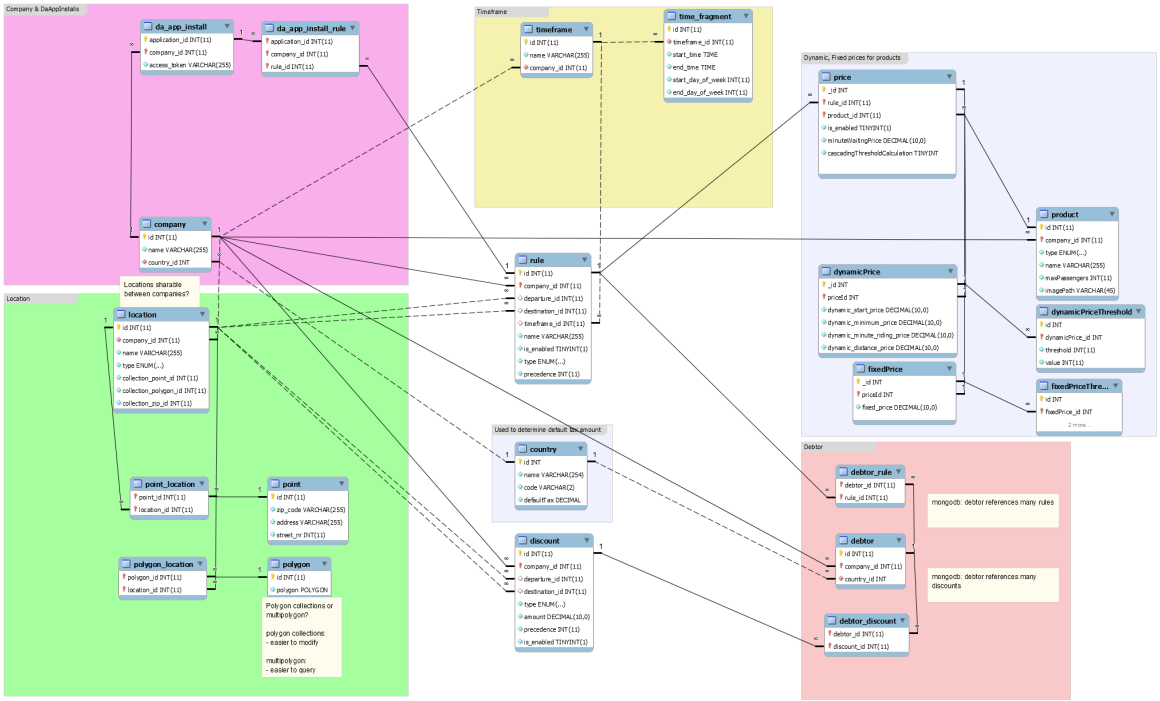


Fig. 3.2 Proposed Database Schema Design

The Company and DaAppInstall entities are fundamental concepts that are found throughout the systems architecture. A company may have multiple dispatch application installations, hence the DaAppInstall abbreviation. To make the schema ready to be implemented, some assumptions were made, for example: the DaAppInstall entity contains a authentication token. This would enable a basic form of authentication and authorization to be present.

A large portion of that data are only relevant for price calculations, and should therefore not be stored in existing databases. Data that are relevant to all systems may include but not be limited to:

1. Company
   (a) id
   (b) name
   (c) taxing / VAT
   (d) Country
   (e) Application
   (f) Debtors
   (g) Rules
   (h) Discounts
   (i) ...
2. Country
   (a) name
   (b) language
   (c) code
   (d) default VAT
   (e) currency
3. Appliation
   (a) id
   (b) name
   (c) Rules
   (d) Discounts

A decision must be made whether company and product information is stored in a shared database that can be accessed by a subset of system architecture components, or whether data should be synchronized in some fashion, or whether there is a possibility that enables dependencies to be eliminated all together.

3.4 Authentication and Authorization

Mobile applications should be able to make requests, just like the portals that are to be developed, but portal users and mobile app users consume microservices in different ways. Mobile apps merely request prices of products, based on the rules that group admins define through the portal. To make sure that only the portal users have the right to mutate their data, users have to be authenticated and authorized within the microservice. Identity management becomes a problem if data duplication is not desired. If a user makes a direct request to the microservice, the credentials have to be compared to user data in a database. In the legacy system architecture, different services implement different authentication methods,

store different pieces of information of different users. One consistent piece of identity in all systems within taxiID is the combination of the company and application installment. These two identifiers describe which resources a user is authorized for. In the legacy system, authorization is achieved by sending extra headers for each crucial piece of information, this is clarified in Appendix A, chapter 3.4. To prevent duplication, the microservice could be connected to the database that is used by the core system. But this makes the microservice less decoupled, and directly contradicts the desire to separate data dependencies as described in the previous section. In Appendix B.3, four proposals were made in which the problem of information dependencies, authentication and authorization were solved using different approaches.

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

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

3.4.1 OAuth 2.0

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

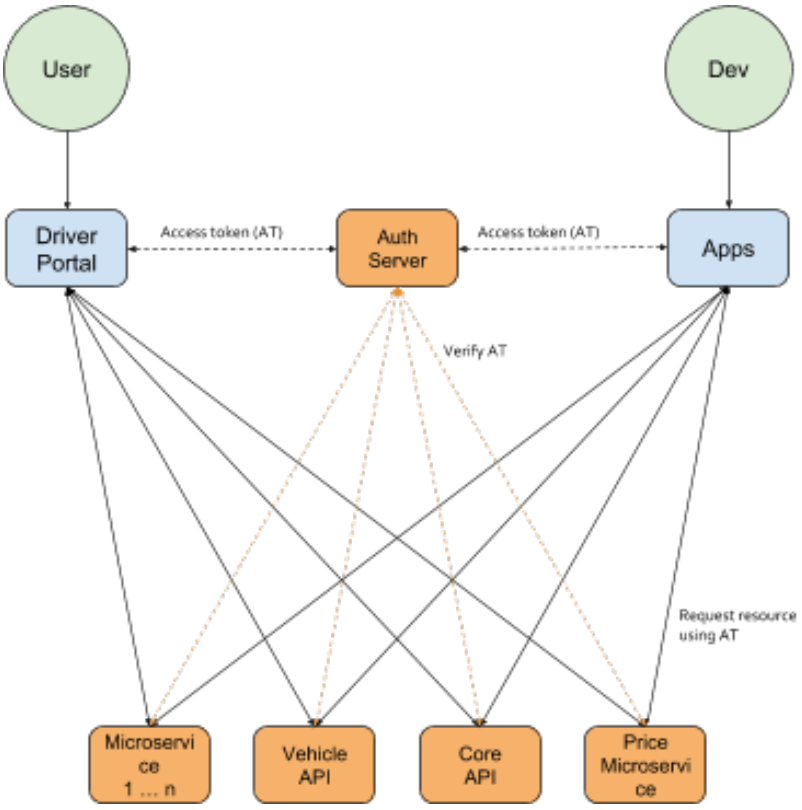


Fig. 3.3 OAuth requests where tokens are verified by Auth Server

3.4.2 JSON Web Tokens

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

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

Listing 3.1 Two user identifiers and registered claim names stored inside the payload of a JSON web token.

The core system signs a token that with a secret that is known by the microservice. The token consists of three parts, separated by a fullstop. The first part (header) of the token

contains information about the hashing algorithm that was used to encrypt the payload. This part is Base64Url encoded. The payload itself contains information stored in JSON format as shown in Listing 3.1. The identity of the user is stored in the payload that can only be revealed by whoever holds the secret with which it was signed. Then the message can be verified using the third part of the token, which is the signature. The verification step prevents tampering with the payload. Claims can be added to the payload as shown in 3.1 to provide information about the token, as explained in [18].

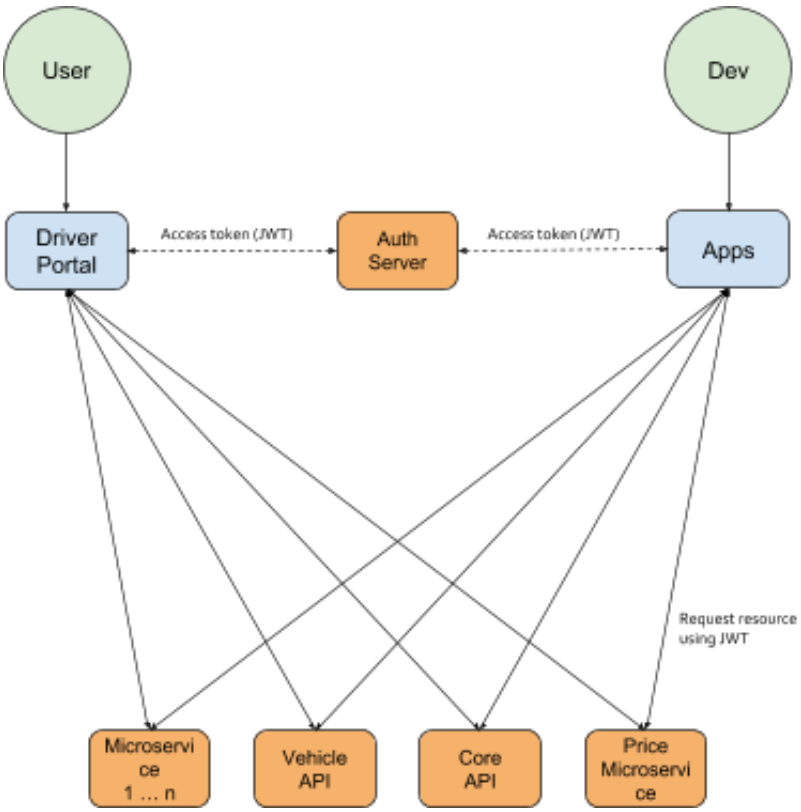


Fig. 3.4 OAuth with stateless JWT token requests

3.4.3 API Gateway

Another common structure that allows services to be used by external agents is the API Gateway. It allows for a central middleware in which authentication and authorization is handled, where the microservices are shielded from public access, and all communication is established through the API Gateway [19]. 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.

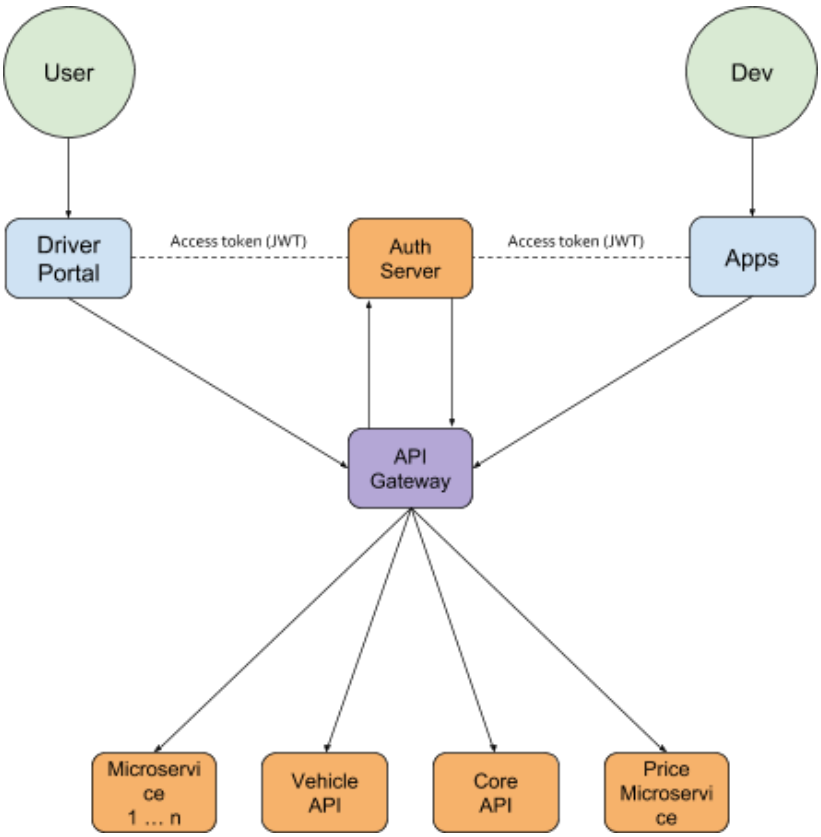


Fig. 3.5 API Gateway

3.5 Technologies

kks32: Rewrite section: why not PHP, MySQL, GraphQL??

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 [20], 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. 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. 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.

3.6 Methods and Techniques

kks32: Expand upon choices: CircleCI, Buddy-Works, Typescript, and what is functional programming?

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 [21] and the assertion library Chai [22]. To further reduce the chances of introducing bugs, some additional techniques could be used.

1. Typescript
2. Functional Programming Paradigm
3. Linting
4. Automated Software Validation

3.7 Conclusion

Taking all the different aspects in this chapter into account, the advised architectural design of TPS comprises of integrated frontend views in each required portal using the associated available technologies, a separate NodeJS microservice with its own MongoDB database, Loopback as a framework to quickly implement functionalities using Typescript and a

combination of OOP and FP, authentication via JWT, automated tests using mocha and chai, and continuous delivery and automated testing using Buddy-Works.

Chapter 4

Trip Price Calculation System

4.1 Introduction

In the previous chapter, information dependencies were discussed. This chapter clarifies which information should comprise a price breakdown to reflect that of the legacy system, what logical flow of information is to be contrived, and how different pieces of information that are stored and processed should restrict the time and space dimensions of a price rule without blurring the straightforwardness of the system.

4.2 The System

The object-oriented programming (OOP) paradigm offers many ways to keep a system structured. Words like 'organized', 'structured', and 'modular' boil down to the same idea: 'a cohesive whole built up of distinct parts'. Good software design has low coupling and high cohesion, meaning that software components should have a high degree of belongingness, and a low degree of dependence in respect of eachother. As stated in the previous chapter, another paradigm was used that compliments price calculation very well called: functional programming (FP). These two paradigms together are capable of providing a modular system that is highly cohesive, and very low coupled. To understand more about the structure of the system, a class diagram visualizes the most important components in 4.1.

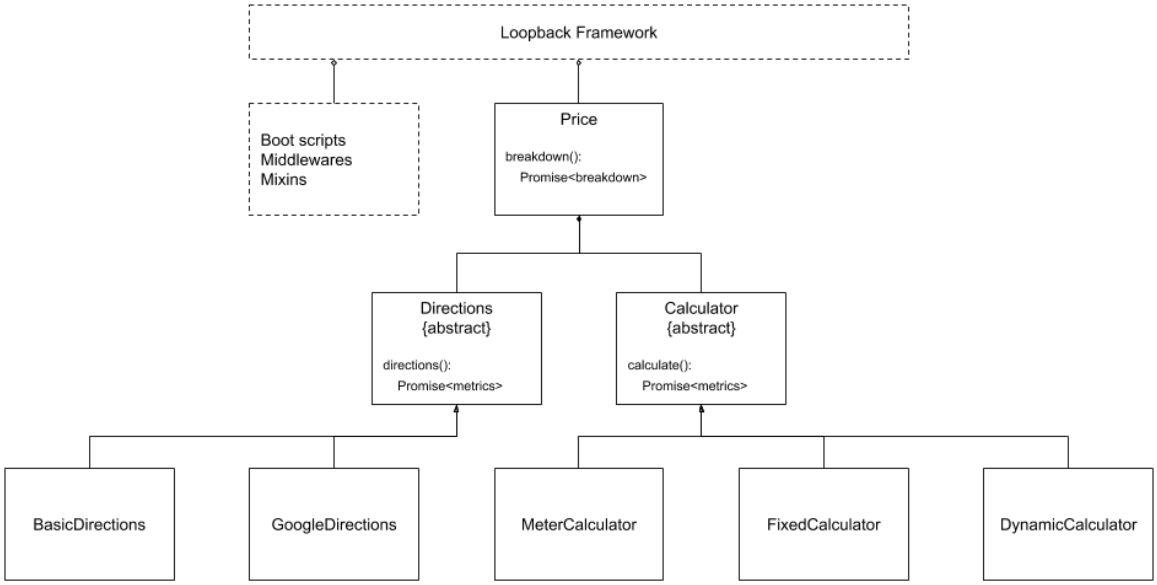


Fig. 4.1 Class diagram.

The Price object is composed of The instance methods shown in the diagram are are used by the associated class that is composed of Within each component, state and mutations are fully encapsulated, leaving only static functions exposed. These functions aim to mutate data in a pure way, meaning that no state is changed outside of the function scope, and that the function is absolutely honest about its parameters and return values. As discussed in the previous chapter, Typescript plays an important role in mixing OOP and FP together.

kks32: Expand on interfaces, static strong types, type hinting, OOP en FP mixen, OOP voor grote structuur, FP voor solide operaties, SOLID, Gang of Four, Loose coupling high cohesion, Async, Strategy pattern,

The database schema desgin as shown in the previous chapter gives an inpression on the different pieces of information required to calculate a price. Such a schema provides a good insight in the relationships that different entities have, but may distract from the actual story that is happening within each calculation. In Figure 4.2 a conceptual model can be seen having association and composition relations in UML notation. This model will be used to refer to throughout this chapter.

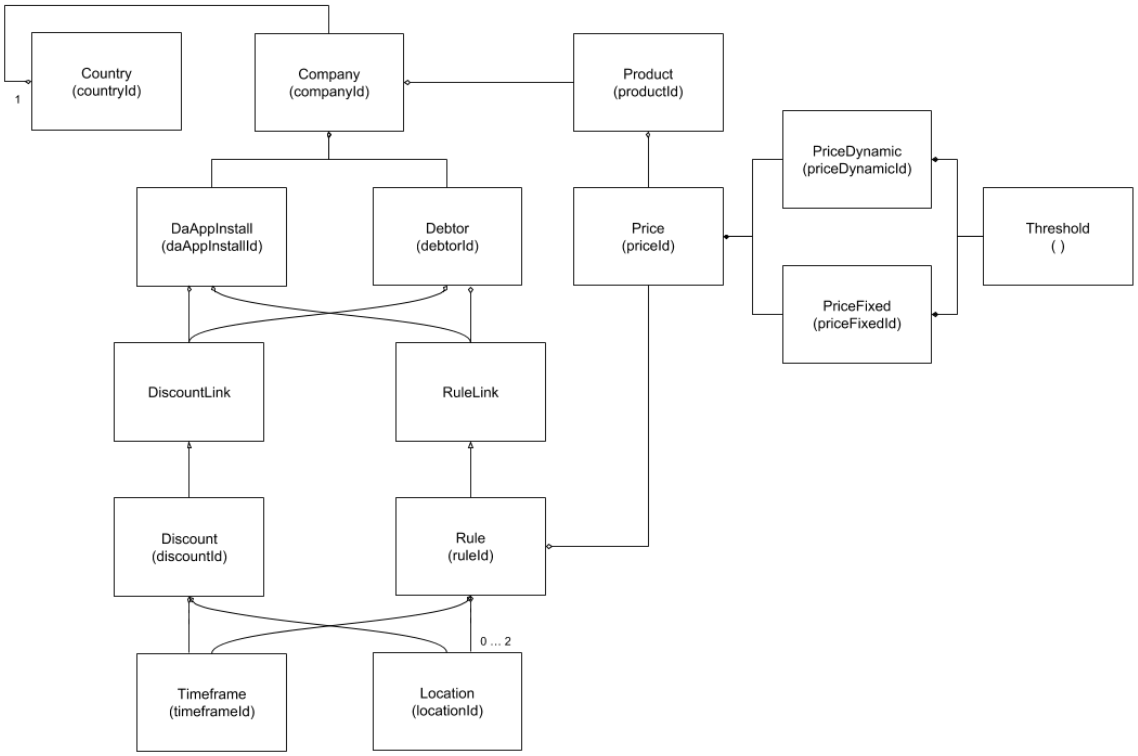


Fig. 4.2 Conceptual data model showing database entity relations.

4.3 Breakdown

The final result of the trip price calculation is a breakdown for every requested product. This chapter starts with the result to make obvious what every step in the process is working towards. 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 4.1 the included tax is part of the breakdown. Is it by mistake or design?

```
[
  {
    "vehicleType": "saloon",
    "maxPassengers": "4",
    "price": {
      "currency": "EUR",
      "total": 850,
      "breakdown": {
        "route": 802,
        "tax": 48,
        "toll": 0,
        "parking": 0,
```

```
        "waiting": 0,
        "discount": 0
      }
    },
    "fixedPrice": "true"
  }
]
```

Listing 4.1 Legacy price breakdown

```
[
  {
    "vehicleType": "estate",
    "maxPassengers": 4,
    "isEstimated": false,
    "price": {
      "breakdown": {
        "route": 8300,
        "toll": 0,
        "parking": 0,
        "waiting": 0,
        "discount": -1650
      },
      "currency": "EUR",
      "total": 6650,
      "tax": {
        "amount": 400,
        "percentage": 6
      }
    }
  },
  ...
]
```

Listing 4.2 Improved 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 4.2. As demonstrated in Appendix B.2, a breakdown is easily constructed in four steps when VAT is included. 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 is chosen to be implemented, where the flag 'fixedPrice' is replaced by the 'isEstimated' flag to clearly reflect its purpose.

4.4 Locations

Locations and timeframes are the big filters that reduce the amount of potential matching rules and discounts based on space and time. The implementation for location queries ...

kks32: Currently implementing this part

4.5 Timeframes

Time plays a role in determining whether a rule has matched. The implementation of this concept should preferably offer enough freedom in the future, and should not be tailored toward one specific entity relation. Being able to reuse the timeframe entity improves maintainability of the system. The requirements state that the user must be able to define a start and end time, the days on which the times are active, and the start and end date of the timeframe. 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.

4.5.1 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.

4.5.2 Bitmap

For this reason, a proposal was made to implement timeframes 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 bitstring format offered the best outcome, as seen in B.6. A timeframe is stored having two ISODates (international standard: ISO 8601), and a bitstring representing the schedule for which the insert statement is shown in Listing 4.3.

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

Listing 4.3 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 bitarray would only allow for 0 and 1 to be used. A bitstring 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 [23]. 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`) };
```

```
21 };
```

Listing 4.4 Opening timeframe.

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.

4.6 The Trip Price Calculation

The directions class provides an interface to retrieve trip related data. The BasicDirections class returns a base case result, while the GoogleDirections class retrieves the data from the Google Directions API. The trip price calculation flow changes drastically when no destination or departure locations are provided in the request body. During development, many requirements were added and removed, handling different cases, returning different results. For this reason, a base case and the google case are defined using a behavioral strategy pattern as described in [24], only using an abstract class instead of an interface. This improves the systems resistence to change. The flowchart in Figure 4.3 shows the on-meter (3) and dynamic and fixed (5) rule queries. Just like the directions, the calculator has its strategies to adapt to other circumstances.

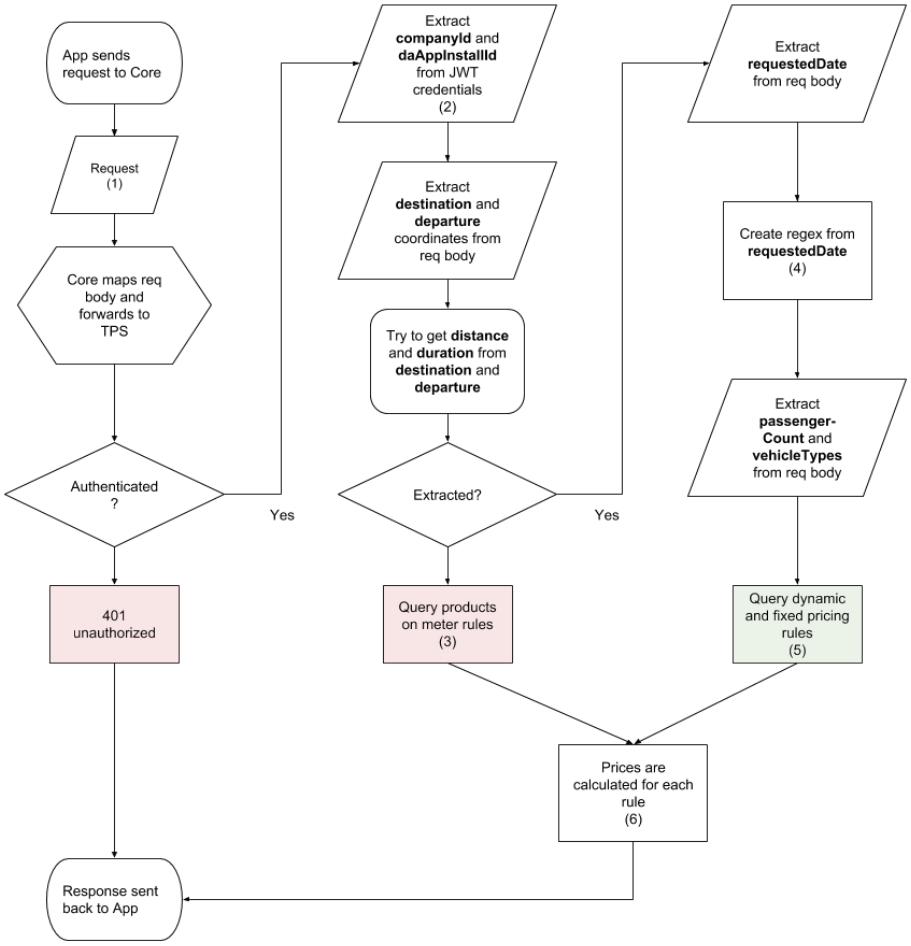


Fig. 4.3 The flow of a trip price calculation.

4.7 Discounts

The price calculation matches rules and discounts separately. Discounts were associated with rules in the legacy system, eliminating the amount of combinations of prices with or without discounts. A discount was either always active, or it was not. Having separation between the two enables users to define other locations and timeframes to both of them.

4.8 Rules

Rules and discounts are queried for each requested product. MongoDB's aggregation framework allows documents to be aggregated in a multi-staged pipeline.

4.8.1 Identification

The first step in the pipeline matches all DaAppInstall entities with the identifiers sent along in the JWT payload: companyId and daAppInstallId.

4.8.2 Links

As shown in Figure 4.2, the DaAppInstall entities have multiple Links; ruleLinks and discountLinks. These links store data in a polymorphic hasManyThrough relation.

4.8.3 Country

Each company has a country, used to determine the default VAT and currency.

4.8.4 Rules

All documents are filtered based on geolocation and timeframes.

4.8.5 Prices

Prices are retrieved for each product that is associated with the matched rule. DynamicPrices and related thresholds, and FixedPrices and related thresholds.

4.8.6 Sorting and Formatting

The final step in the aggregate sorts and formats the results so that the Price class can calculate a price breakdown for each result.

4.9 Price Calculation Types

4.9.1 Dynamic

4.9.2 Fixed

4.9.3 Meter

4.10 Threshold Calculations

4.11 Conclusion

Chapter 5

Proposed Portal Solution

5.1 Introduction

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

5.2 Required Views

5.2.1 Entities

- products - rules - discounts - locations - apps
    - timeframes - location picker

5.2.2 Hierarchy

- overview pages - detail pages - composite pages

5.3 Methods and Techniques

5.3.1 Pricing Rules

5.3.2 Timeframes

5.4

5.5

How can complex pricing rules be communicated through the UI?

Which views are essential? In what way can visual hierarchy guide the user through processes naturally? How should complex elements impacting price calculations be communicated to the user through UI components?

Chapter 6

Realization

6.1 Introduction

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

6.2 Methods and Techniques

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

6.3 Sprint 1 - Dynamic Price Calculations

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

6.4 Sprint 2 - Authentication and Authorization

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

6.5 Sprint 3 - Setting up the Portal

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

6.6 Sprint 4 - Expanding the Portal

Feedback was given by the product owner after each sprint, resulting into new requirements and modifications to requirements. A functionality was required that enabled the user to sort pricing rules and special rates (discounts), by dragging the rows in a table to the correct positions. Whenever a priority changed, all subsequent rows needed to be updated to maintain a consistent prioritized list without duplicates. Another requirement would enable products to

be returned in the breakdown as 'on-meter' results. This meant that, whenever a destination or departure location was undefined, the system would return products without a price, so that the apps could assign a price later, but would still be aware of the available products. A view was added that displayed all apps of a company, and a detail page was added in which rules and discounts could be associated with those apps. This detail page was created in three iterations. The titles and labels of all pages were replaced by references to the localization api for internationalization as seen in Figure 3.1. Timeframe components were added to multiple views, in which hours per week could be specified between two dates.

6.7 Sprint 5 - Expanding the Portal

Thresholds, just like prioritized rules and discounts, had to be ordered, as they were independently embedded in multiple price entities that had 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 were allowed, and no empty price values were allowed inside the thresholds. Thresholds could independently be removed and added, where in the last case values would be copied from the last to the newly added threshold so that the user would not need to manually insert all values. The timeframes were expanded so that either hours per week were used to determine whether a rule should be triggered, or a time could be set to further restrict the begin and end date of the timeframes. Price estimations were added as a setting on all relations between all entities and rules, or discounts. Portal authentication was implemented so that a JWT was requested at the core system, the token stored in localstorage and used to communicate with TPS. The automatic call to generate synchronized company data was enabled for the core system, such that every time a company was created in the core system, it was also created in TPS. A locations overview was added that displayed a list of areas and points. The detail page for each location shows a map with the shape or point displayed.

6.8 Sprint 6 - Locations

TBA

6.9 Result

Chapter 7

Conclusion

A generic location-based price calculation system be implemented by separating ...

Chapter 8

Recommendations

8.1 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.

8.2 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.

8.3 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

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

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.

8.3.1 Authentication section 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.

8.4 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.

8.4.1 section 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.

References

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

[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] 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

[4] (2004) Geographic information – simple feature access – part 1: Common architecture. [Online]. Available: https://www.iso.org/standard/40114.html

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

[6] (2016) what3words - restful api. [Online]. Available: https://docs.what3words.com/api/v2/#autosuggest-params

[7] (2018) Postgis 2.4.5dev manual. [Online]. Available: https://postgis.net/docs/manual-2.4/using_postgis_dbmanagement.html#PostGIS_GeographyVSGeometry

[8] (2018) Mysql 5.7 reference manual - geometry class. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/gis-class-geometry.html

[9] (2018) Geospatial query operators — mongodb manual 3.6. [Online]. Available: https://docs.mongodb.com/manual/reference/operator/query-geospatial/

[10] 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

[11] (2015) Geospatial performance improvements in mongodb 3.2. [Online]. Available: https://www.mongodb.com/blog/post/geospatial-performance-improvements-in-mongodb-3-2

[12] 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

[13] (2011) Iso/iec 25010:2011. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en

[14] 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

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

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

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

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

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

[20] (2018) Loopback 3.x - loopback documentation. [Online]. Available: https://loopback.io/doc/en/lb3

[21] (2018) Mocha - the fun, simple, flexible javascript test framework. [Online]. Available: https://mochajs.org

[22] (2018) Chai assertion library. [Online]. Available: http://chaijs.com

[23] (2015) Mongodb evaluation query operators. [Online]. Available: docs.mongodb.com/manual/reference/operator/query/regex/#index-use

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

List of figures

List of tables

Appendix A

Pregame

Pregame

Phase I - Pregame

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

betreft Afstudeerstage bij TaxiID

jaar 2018

Author: Stefan Schenk
500600679

TaxiID
05-02-2018 te Medemblik

Tutor: Willem Brouwer

Hogeschool van Amsterdam
Software Engineering

Index

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

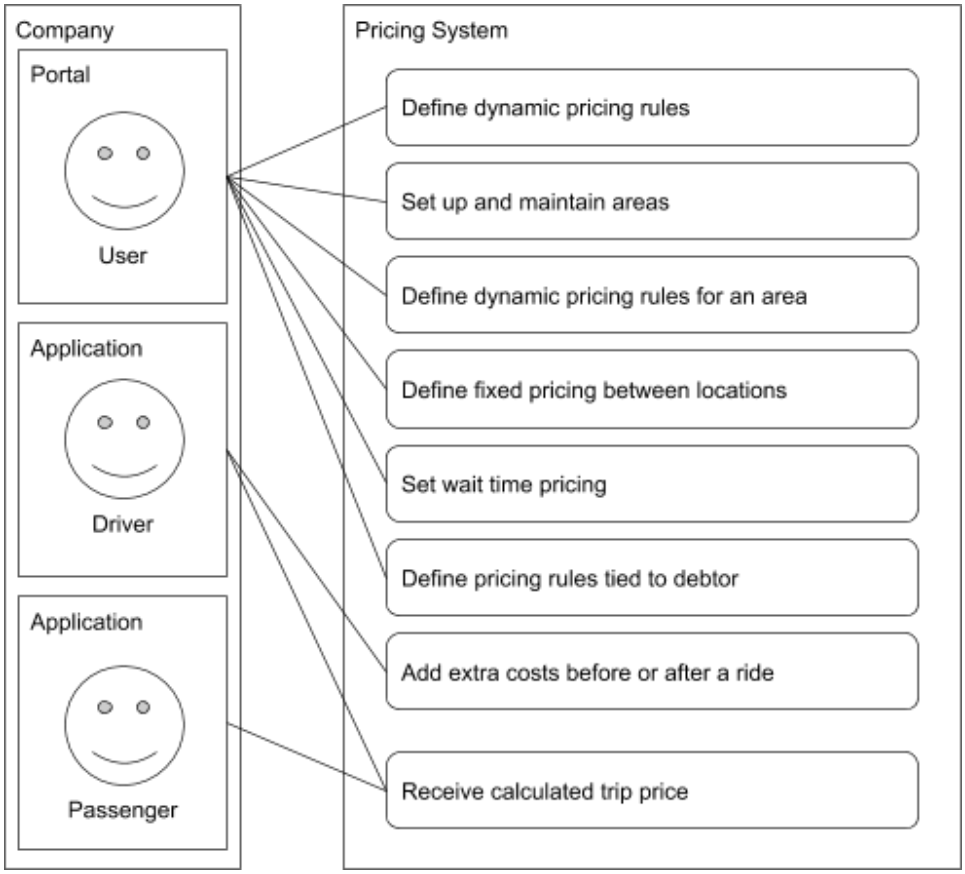


Image 2.4.1 - Use case diagram.

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.

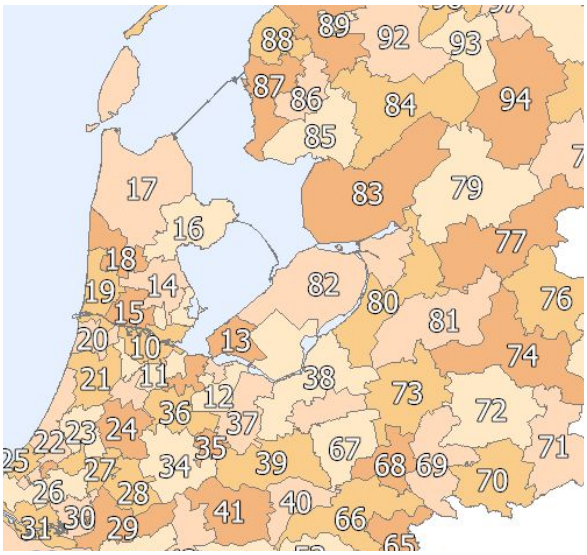
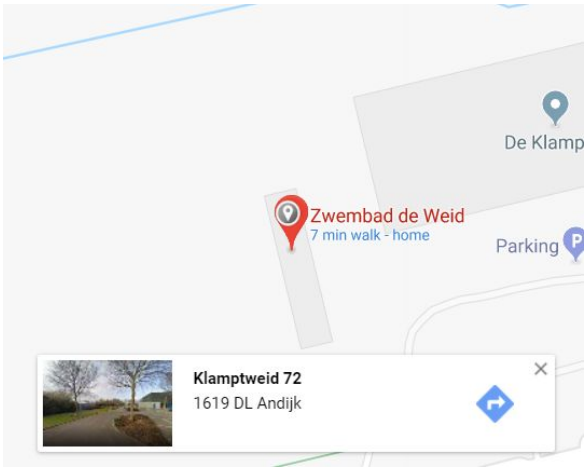
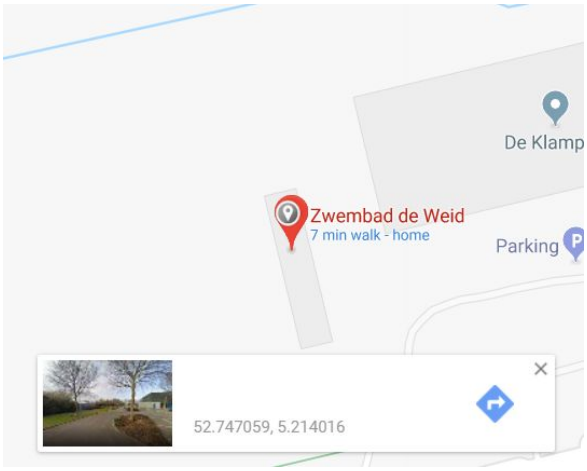
3.2. Functional Requirements

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.

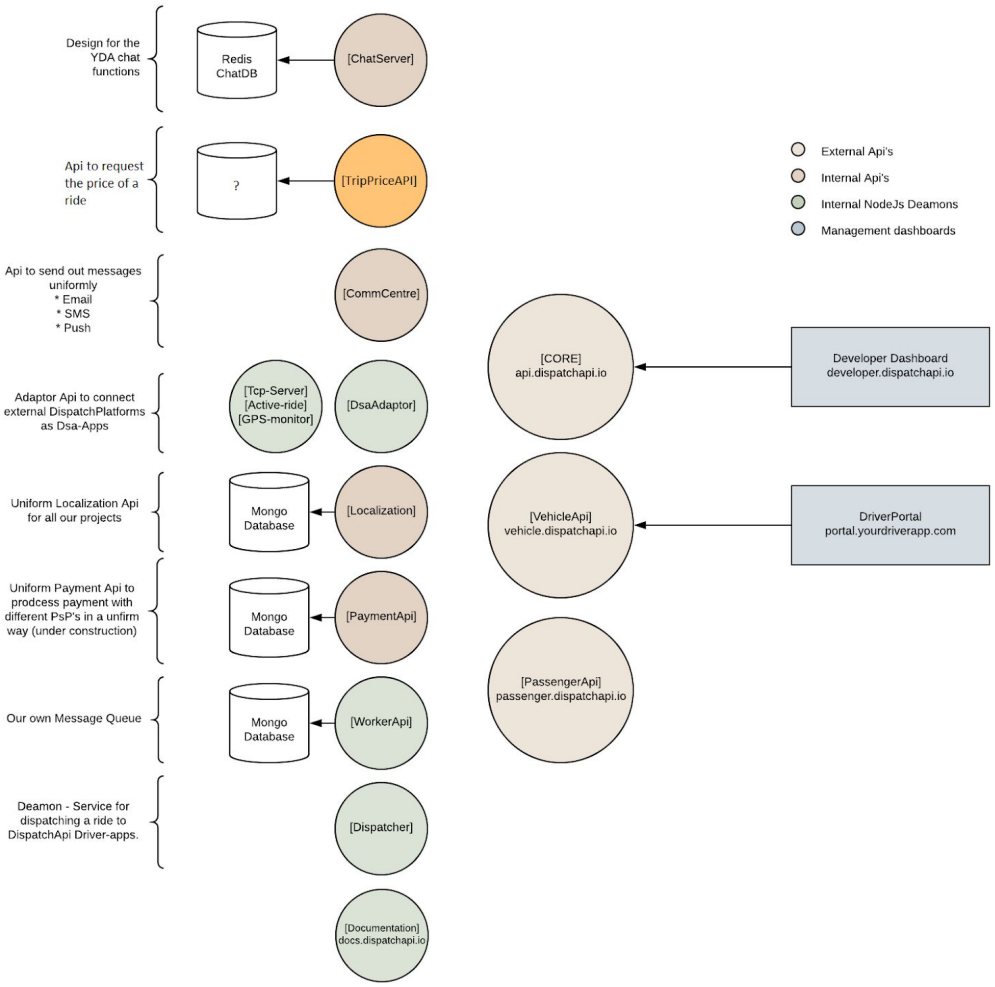


Image 3.3.1 - Current system architecture of the dispatch api.

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.

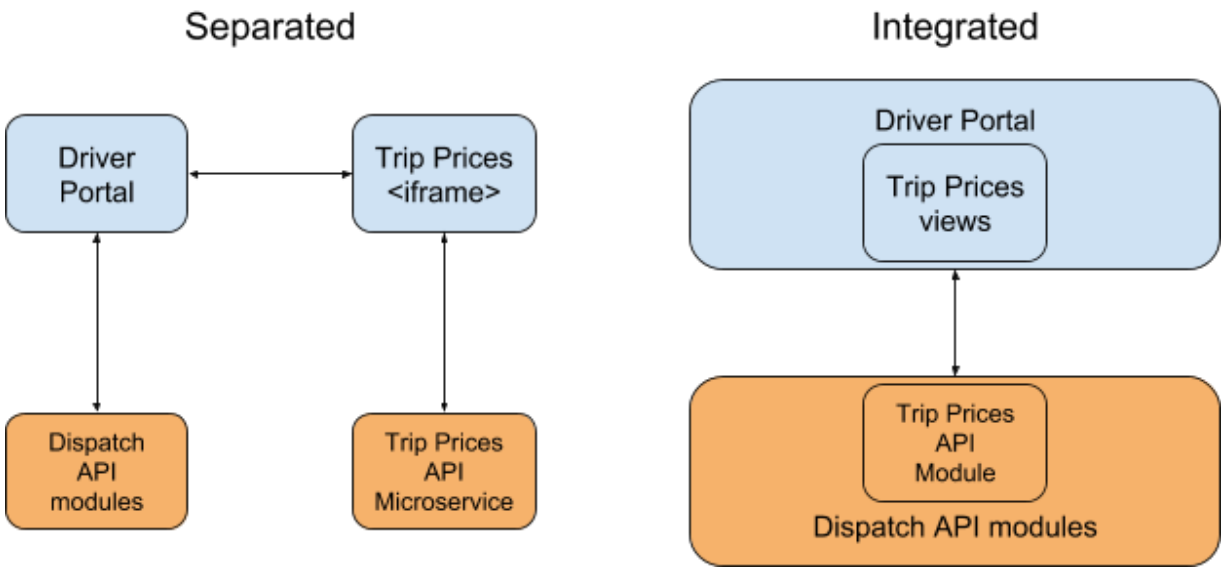


Image 3.3.2 - Separated and integrated frontend and or backend.

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:

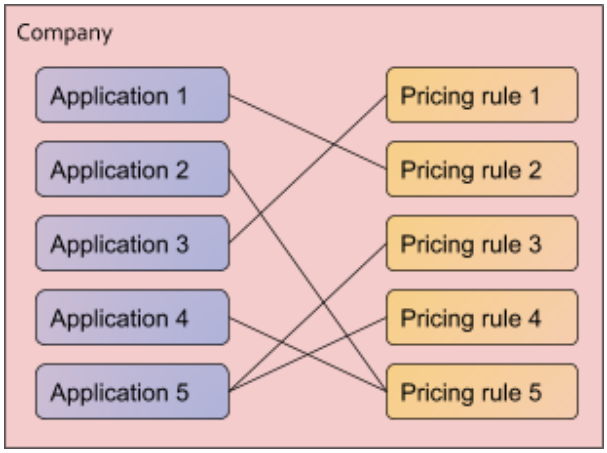


Image 4.1.1 - Company with applications and pricing rules.

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.

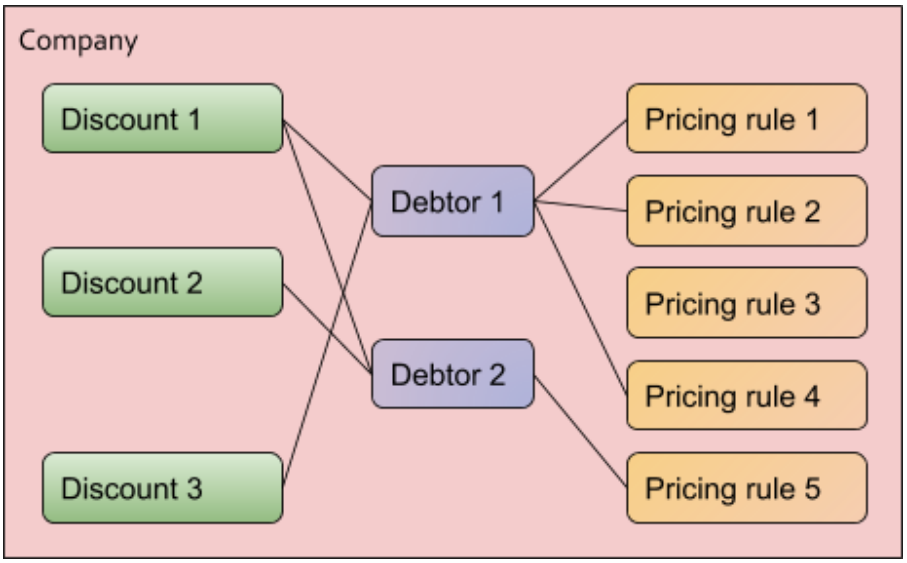


Image 4.2.1 - Debtors and the relation with pricing rules & discounts.

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 4.1.1 - Pros and Cons of separation.

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

4.4. Authentication and Authorization

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

4.4.1. Proposal oauth 2.0 refactor

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

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

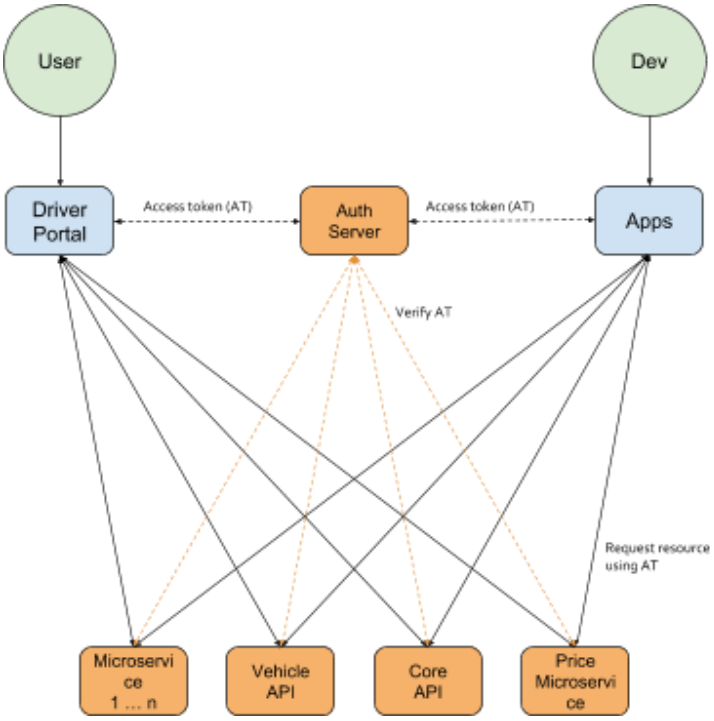


Image 4.4.1.1 - OAuth requests where tokens are verified by Auth Server.

4.4.2. Jwt token format proposal

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

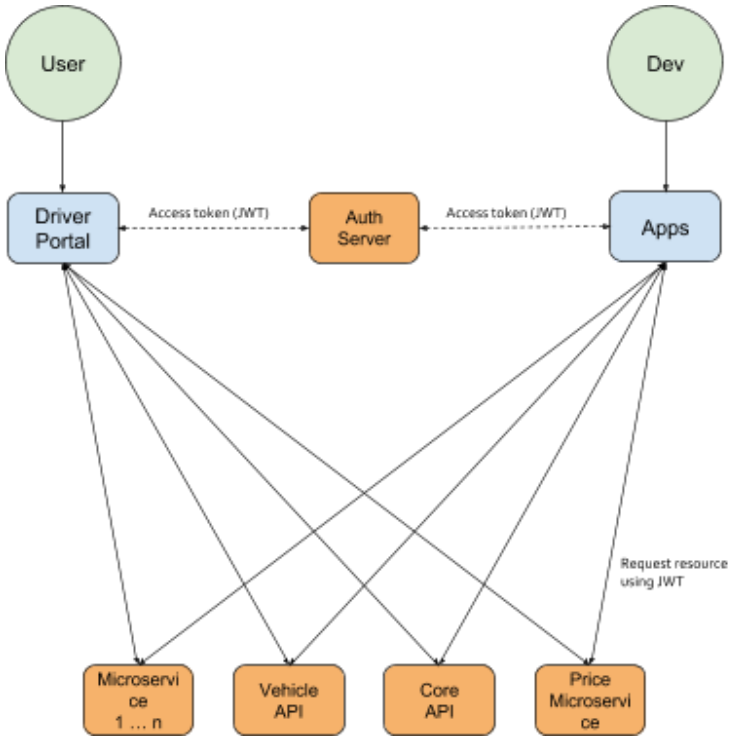


Image 4.4.2.1 -OAuth with stateless JWT token requests.

4.4.3. Proposal API Gateway

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

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

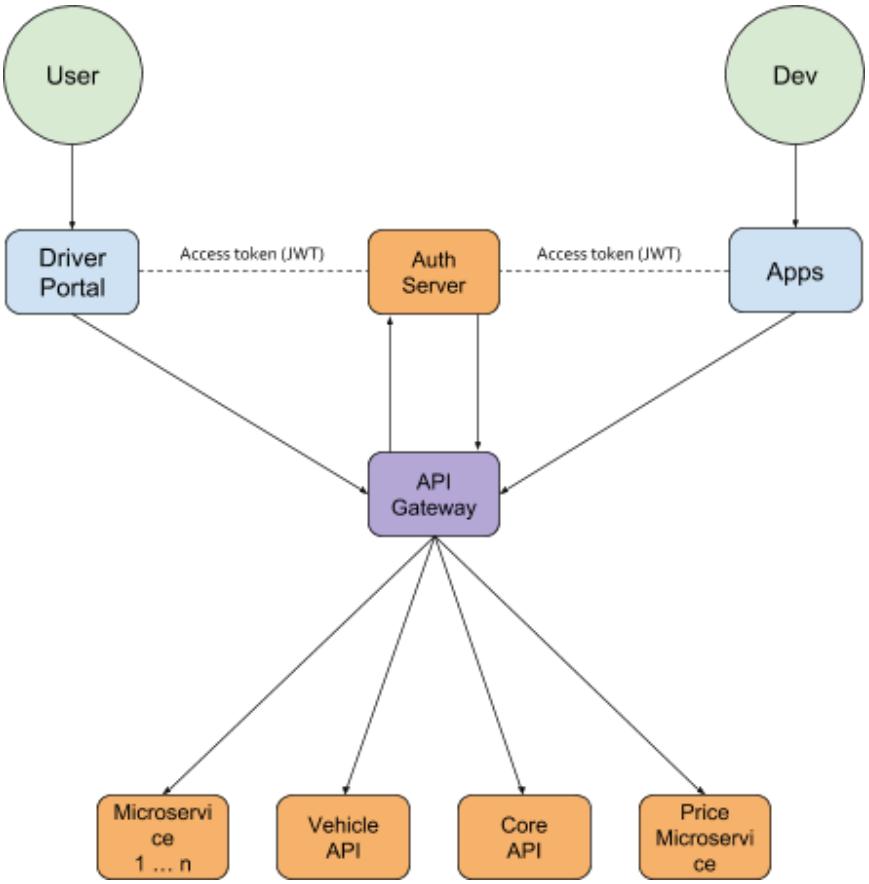


Image 4.4.3.1 - API Gateway.

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

4.5. Database

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

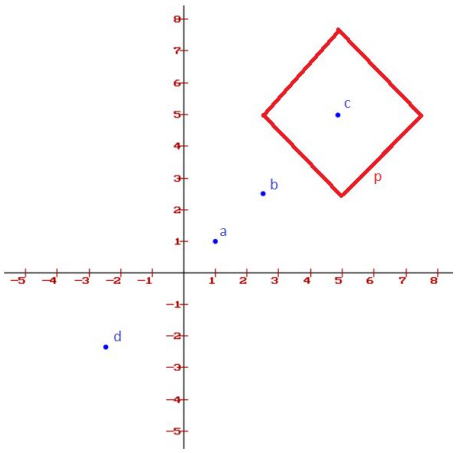


Image 4.5.1 - Four Points and one Polygon p containing Point c.

4.5.1. OpenGIS Compatible databases

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

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

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


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

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

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

```
// 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

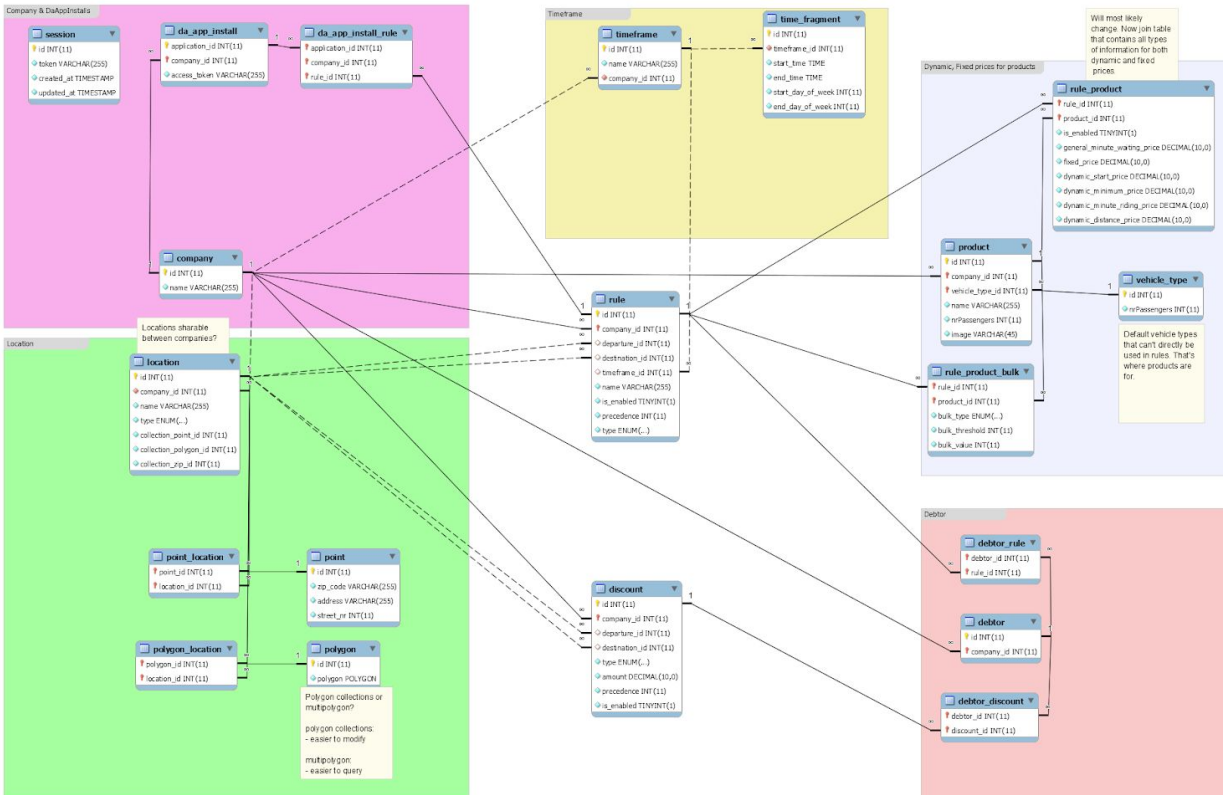


Image 4.7.1.1 - Rough schema for a relational database.

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

B.1 Sprint 1 - review

Sprint 1
Dynamic Price Calculation

with companyId, vehicleTypes, maxPassengers, enabled pricing rules, ordered by precedence, using distance and duration based pricing

Fake Data

Faker

Random data is inserted in the database using data fixtures

```
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.*}"
```

_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...

Price Calculation

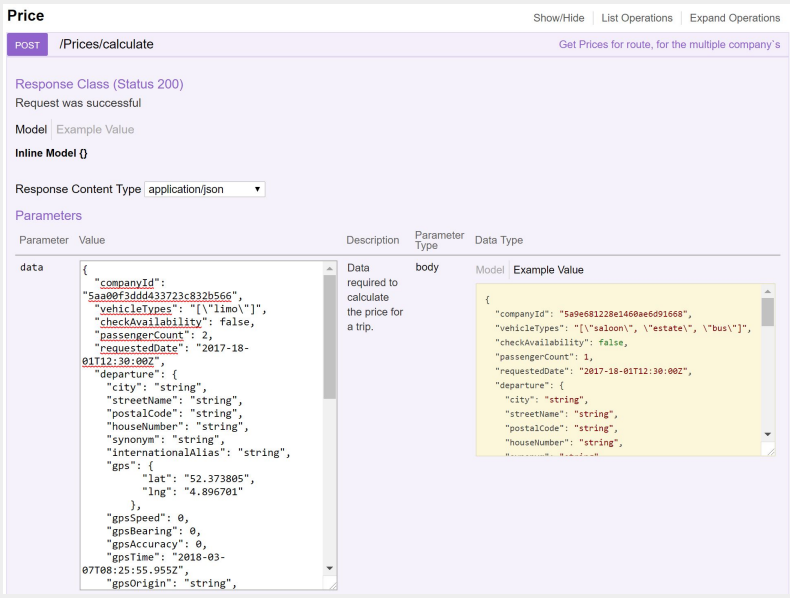
Step 1 - PassengerApp sends request to TPS

The following 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 vehicleType.

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 being used currently.

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 not used yet:

4. departure
5. destination
6. pickupTime

```
const aggregateQuery = () => {
  Product.dataSource.connector.db.collection('Product')
    .aggregate([
      {
        $match: {
          // for a given company
          companyId: ObjectId(body.companyId),
          // vehicles requested must match the product vehicle type
          type: { $in: JSON.parse(body.vehicleTypes) },
          // maxPassengers is bigger or equal to passengerCount
          maxPassengers: { $gte: body.passengerCount }
        }
      },
      {
        $lookup: {
          from: "ProductPricing",
          localField: "_id",
          foreignField: "productId",
          as: "productPricings"
        }
      },
      {
        $unwind: {
          path: "$productPricings",
          preserveNullAndEmptyArrays: false
        },
      },
      {
        $match: {
          // product for given rule is enabled
          "productPricings.isEnabled": true
        }
      },
      {
        $lookup: {
          from: "PricingRule",
          localField: "productPricings.pricingRuleId",
          foreignField: "_id",
          as: "pricingRules"
        }
      },
      {
        $unwind: {
          path: "$pricingRules",
          preserveNullAndEmptyArrays: true
        }
      },
      {
        $match: {
          // rule is enabled
          "pricingRules.isEnabled": true
        }
      },
      {
        $sort: {
          "pricingRules.precedence": -1,
          // should be -1 later, because fixed should go first
          "pricingRules.type": 1
        }
      },
      {
        $limit: 1
      }
    ]).toArray((err, data) => {
```

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.

Price formula

```
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

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 used distance metric)

Step 5 - Sending back the response

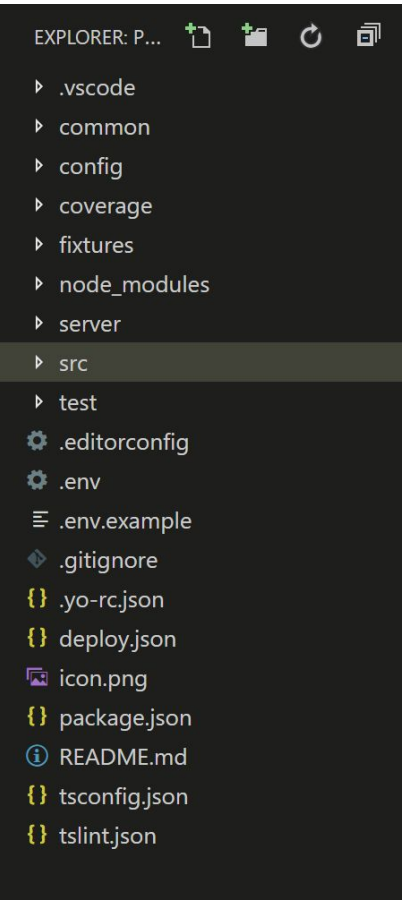
When all the prices have been calculated (for each vehicle type), the response is sent back to the PassengerApp.

Response

each vehicle type has none or 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



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: 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.

   These tests may be removed when it stands in the way of CI.

Tests: debugging

1. Debug flag
   Set 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,
maxPassengers, ...) } to deeply equal { Object (route, tax, ...)
}
      + 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 5x       culator.validPricingOrError(pricing);
22 5x       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

Price Breakdown

breakdown of price properties

Preferred layout:

```
"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
  }
}
```

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

```
"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

```
"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

End

```
"price": {
  "breakdown": {
    "discount": -10.5468
    "parking": 1.88
    "route": 61.1
    "tax": 3.8148
    "toll": 4.7
    "waiting": 2.632
  }
  "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 = [taxperc * prop : prop in breakdown]
   Mutate prop = [prop - tax : prop in breakdown]

✔ `[-11.22, 2, 65, 5, 2.8].map(v => v - v*0.06)`
`= [-10.5468, 1.88, 61.1, 4.7, 2.632]`

```
"price": {
  "breakdown": {
    "discount": -10.5468
    "parking": 1.88
    "route": 61.1
    "tax": 3.8148
    "toll": 4.7
    "waiting": 2.632
  }
  "currency": "EUR"
  "total": 63.58
}
```


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 + parking + route + tax + toll + waiting

✔ -10.5468+1.88+61.1+3.8148+4.7+2.632 = 63.58


```
"price": {
  "breakdown": {
    "discount": -10.5468
    "parking": 1.88
    "route": 61.1
    "tax": 3.8148
    "toll": 4.7
    "waiting": 2.632
  }
  "currency": "EUR"
  "total": 63.5
}
```

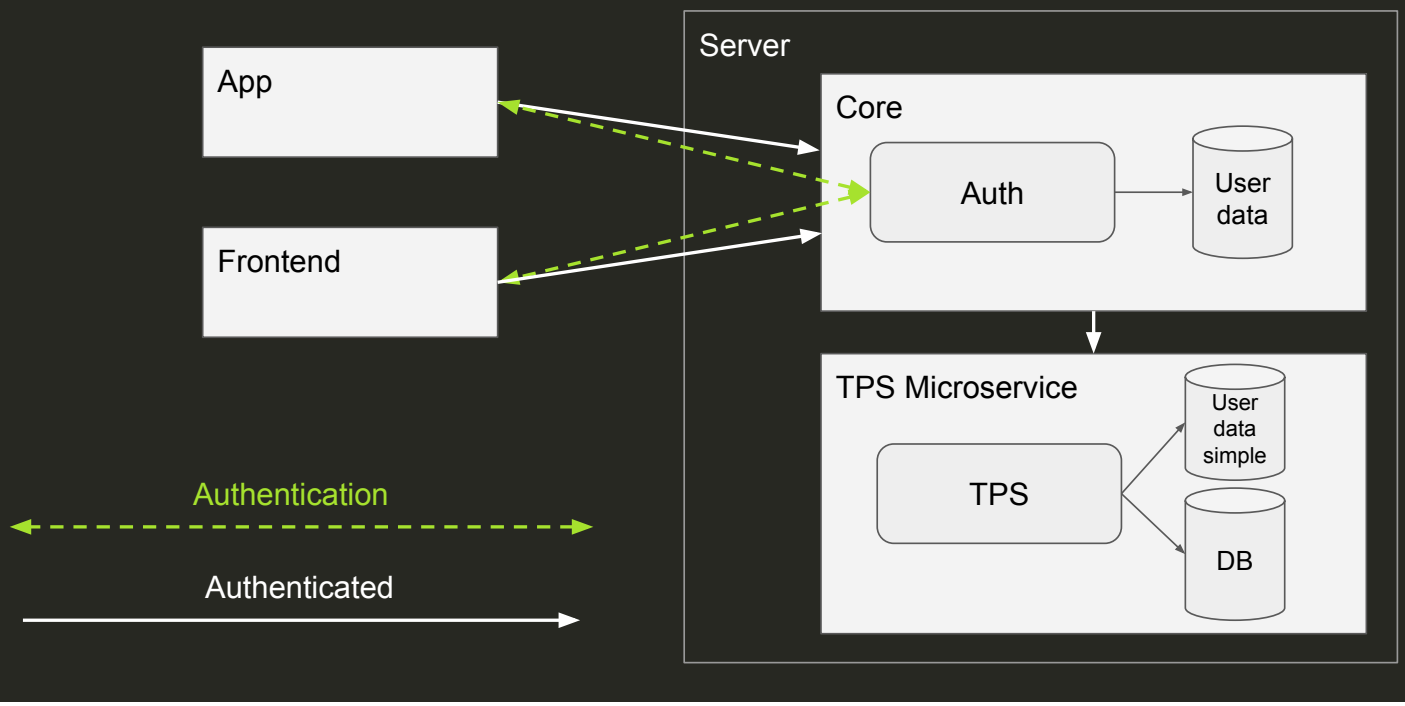

1. Parking, toll, waiting and route prices are calculated
2. A subtotal = parking + route + toll + waiting, is calculated
3. Discount..
   a. Is stated as a positive or negative fixed amount
   b. Is negative or positive percentage of subtotal
4. Tax = (discount + subtotal) * tax_percentage
5. Total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

B.3 Sprint 2 - authentication

Authentication and identity management in a microservice

delegation of responsibility

Preferred layout:



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 Pregame document

Aspects of authentication 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

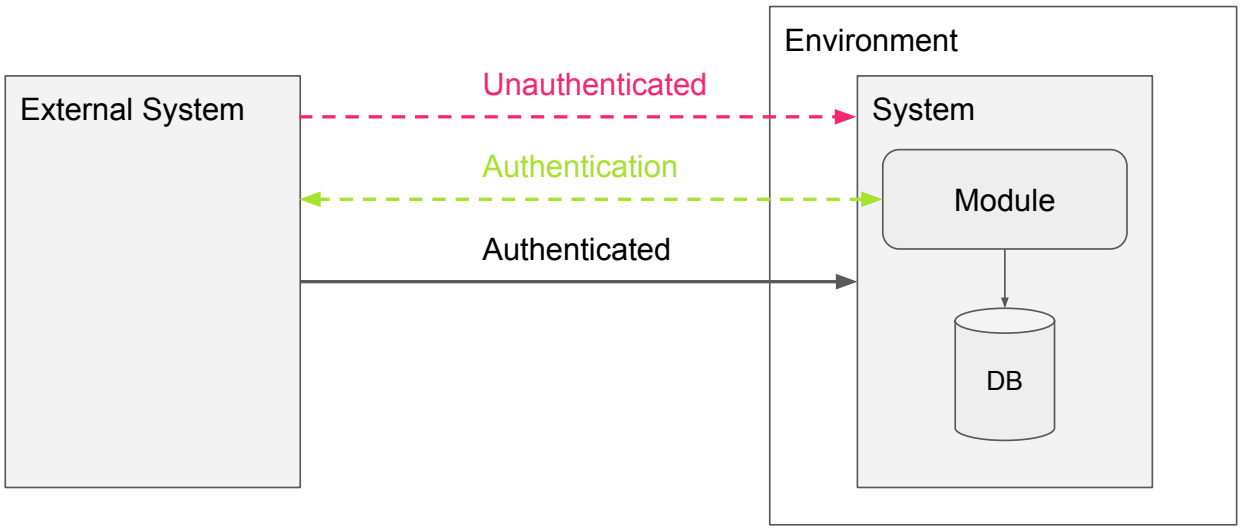
Examples

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

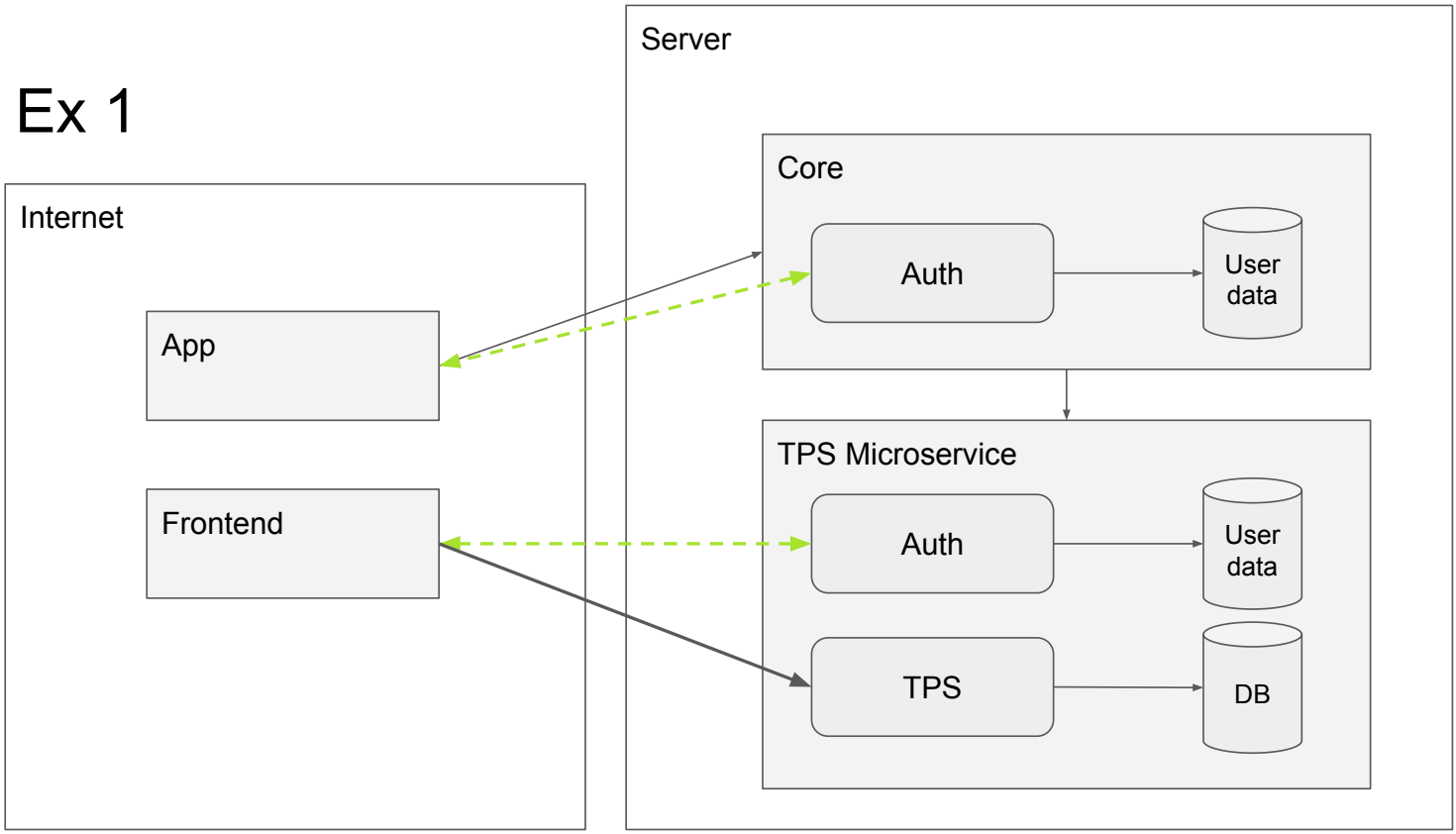
The four aspects are discussed after each figure:

- Responsibility
- Locality
- Authorization
- Statefulness

Symbols used in examples



Ex 1



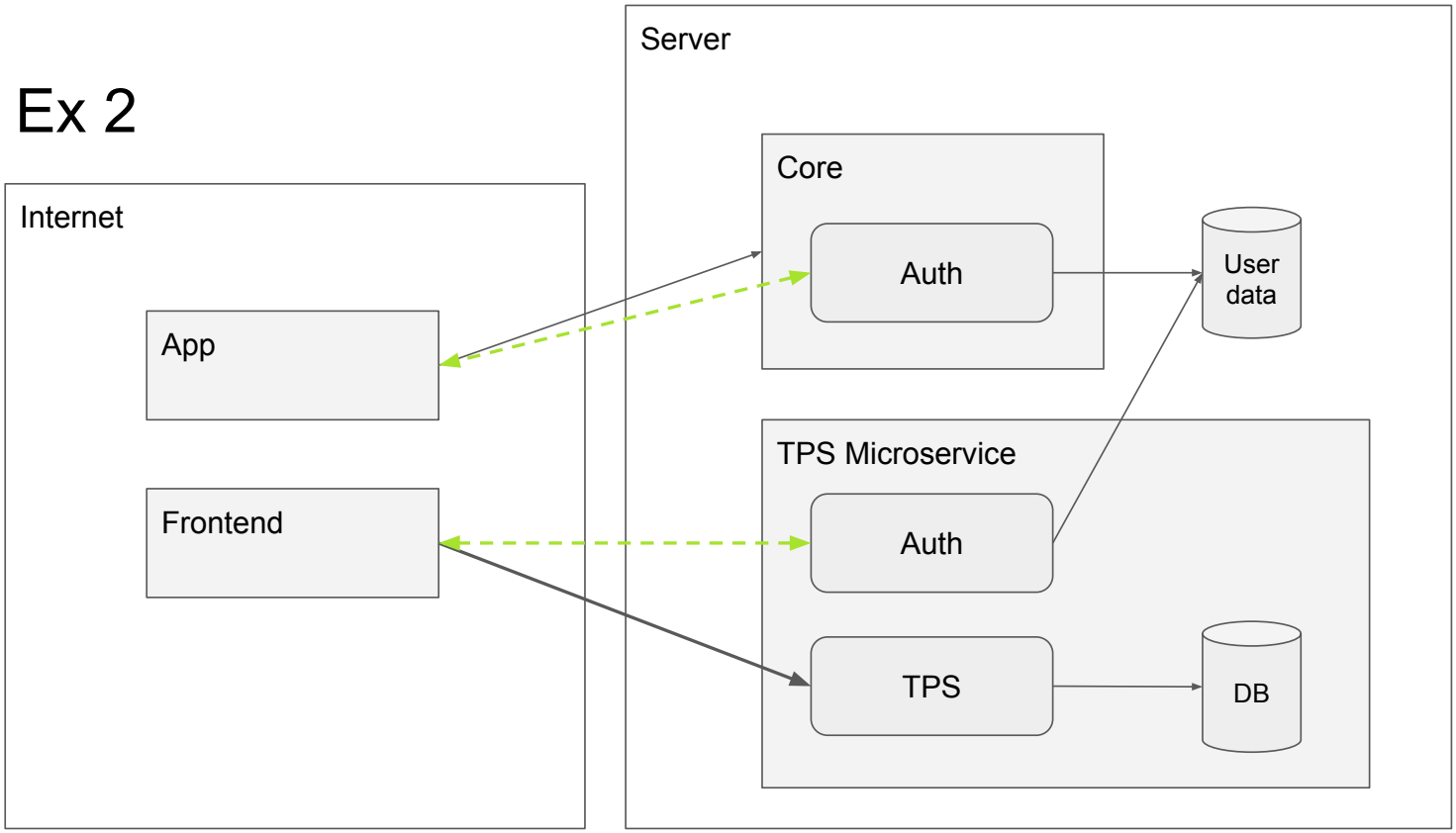
Ex 1 - Aspects

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

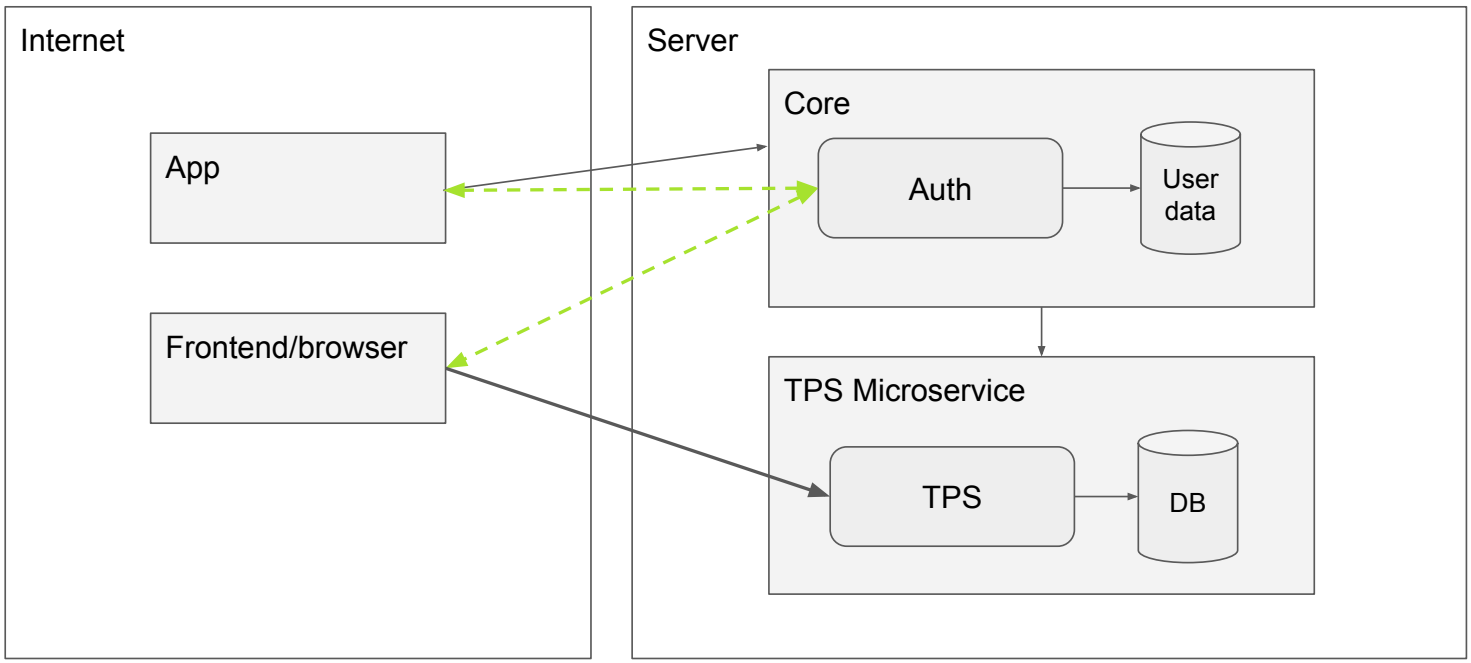


Ex 2 - Aspects

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



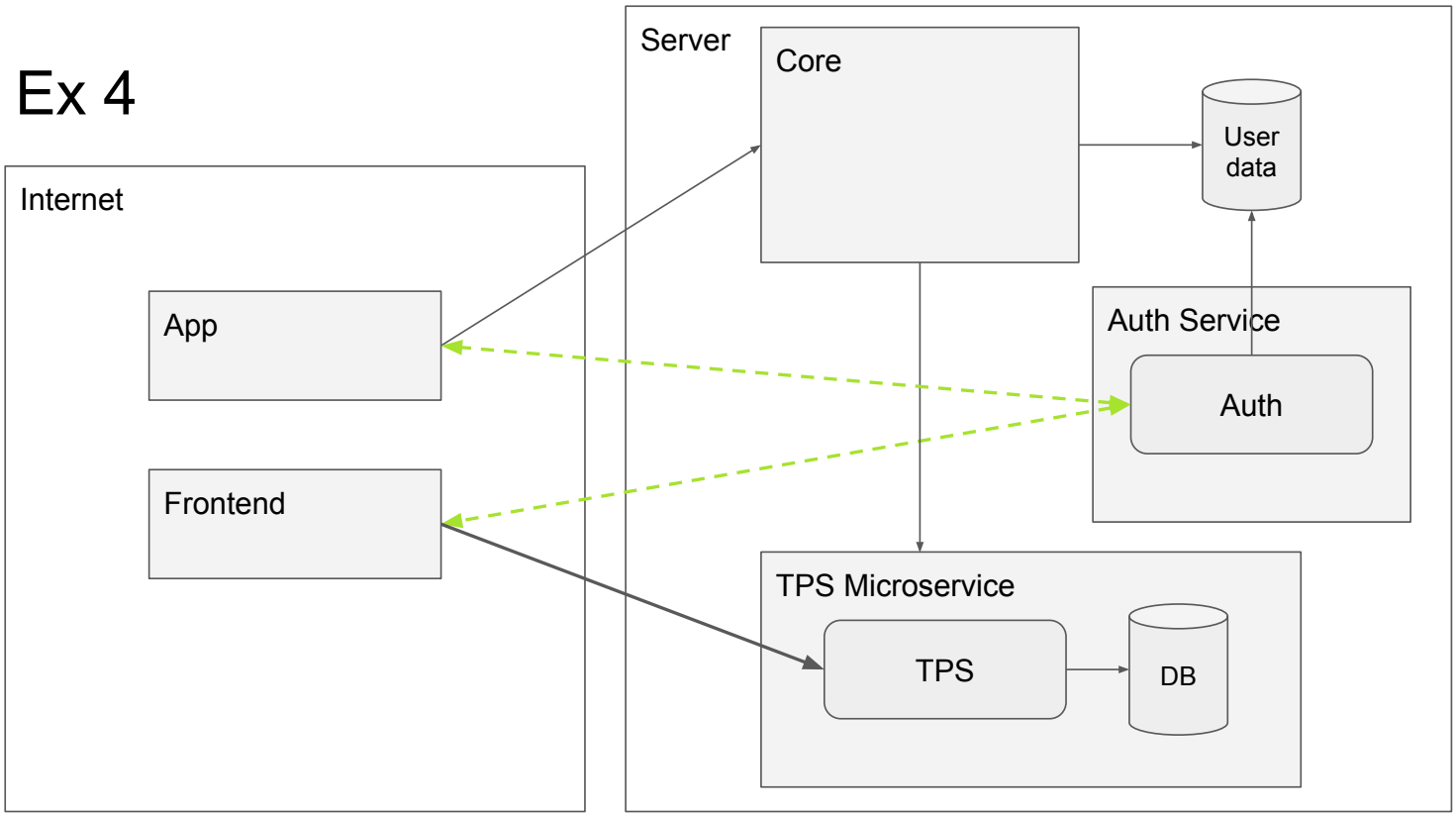
Ex 3 - Aspects

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 in the future, more systems have to make use of the microservice, 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.

Things to keep in mind with JWT

- 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 - review

Sprint 2
Dynamic Price Calculation

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

Intro

A responsible person should think about these things some time in the future.

Room for improvement

Some implemented features could be improved to perform better, use less resources, be more readable, be more testable, etc.

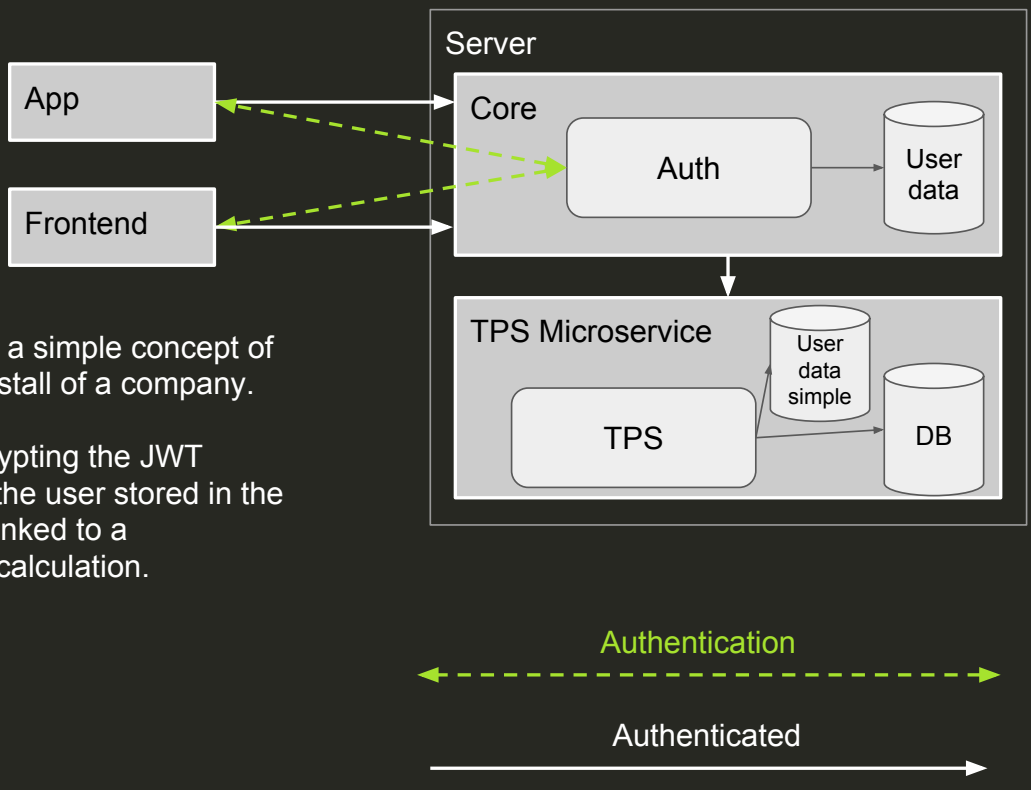
Name	Improvement	Severity	File location
Query to fetch pricing rules for multiple products	The query is run once for every vehicle (product), because no solution was found quickly enough to run it as once single query.	low	/common/price.js
Loopback models are not Typescript ready	No JS files should be used as types can't be checked in transition from Loopback model (dynamic content) to TS files.	average	/common/models/*
Tests include environment specific checks	A strategy should be applied to make these tests work in non-production environments.	low	/test
Authentication is not integrated with Loopback	A middleware checks the headers for a token, bypassing the built in Loopback API Explorer authentication field. Authentication must manually be disabled currently to be able to work with the Explorer. Integration would be an improvement.	low	/src/middleware/authentication.ts
Exceptions are thrown	In some edge cases, exceptions are thrown. This should be looked at by someone who knows what the outcome should be in these cases.	high	**/*.ts (search 'throw')

Authentication

Authentication

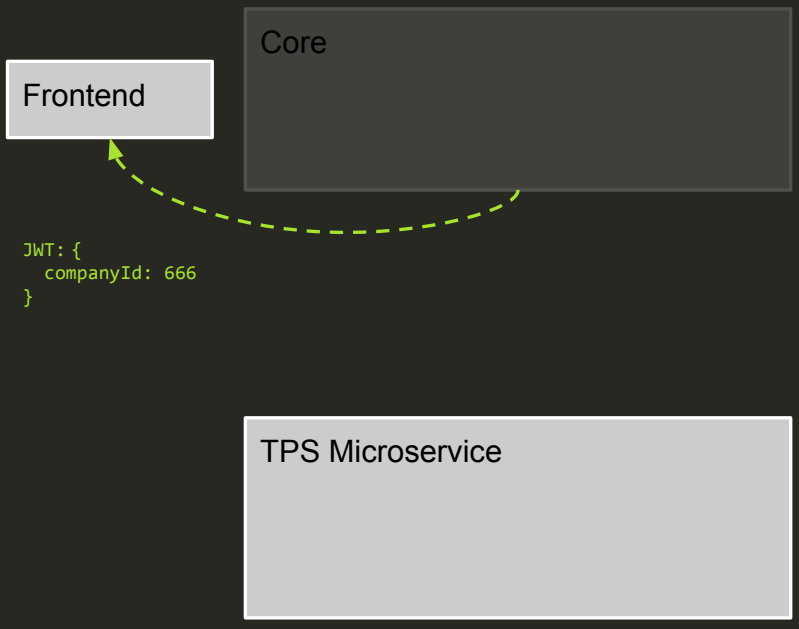
Applications communicate with the core system which provides a JWT that includes important identity information in the token payload. The TPS microservice has a simple concept of a User that references the DaAppInstall of a company.

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

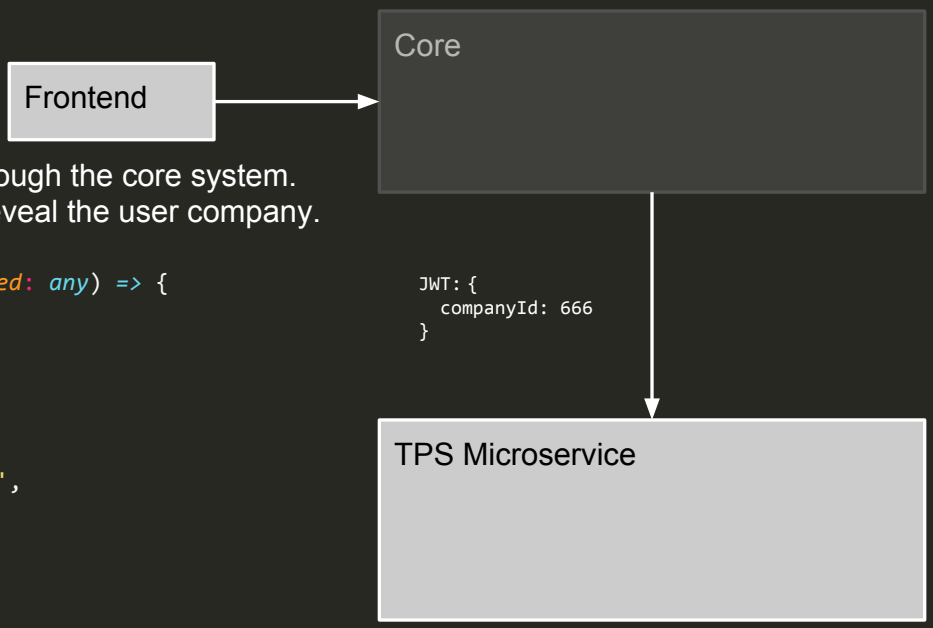
Core signs JWT that
has the identity of the user
contained in the payload, e.g.:




```
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( )

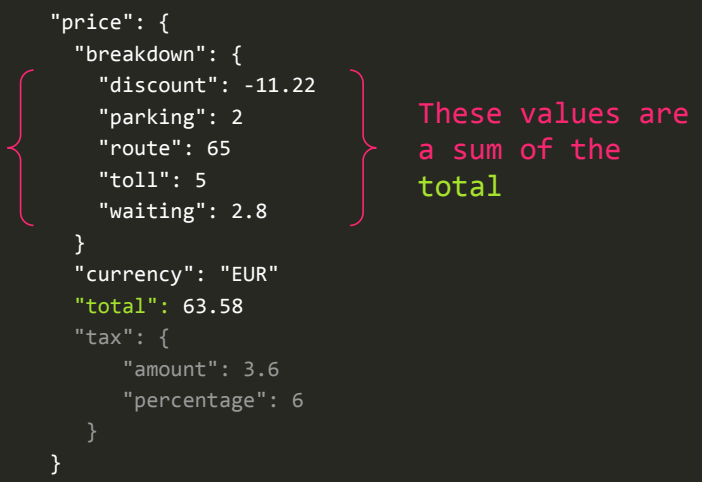
```
/**
 * Authentication middleware.
 */
export const auth = (req: Request, res: Response, next: NextFunction) => {

 // 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.
6. The discussion is found here.

Improved Breakdown

Improved Breakdown



Improved Breakdown

```
"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 here.

VAT

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 here.

Established formula used:

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

Discounts

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

A discount can be negative or positive.

A discount can be disabled or enabled.

A discount can be a fixed amount or percentage.

A discount is calculated and is part of the breakdown total.

Discounts are not constrained by location or timeframes yet.

Cascading Threshold Calculations

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

...
```

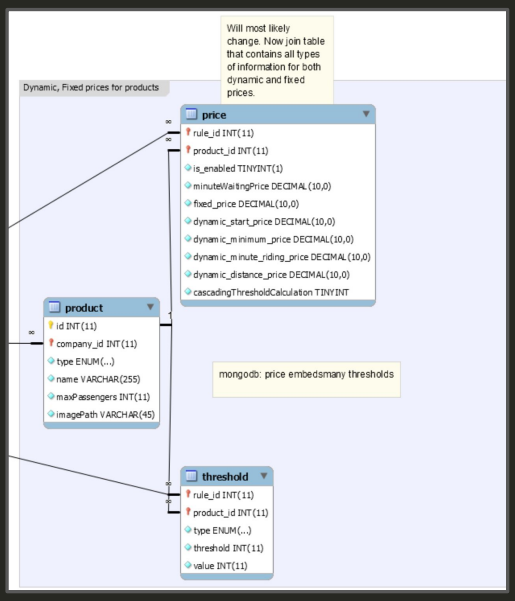
```
...

    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

Refactors: DB Schema

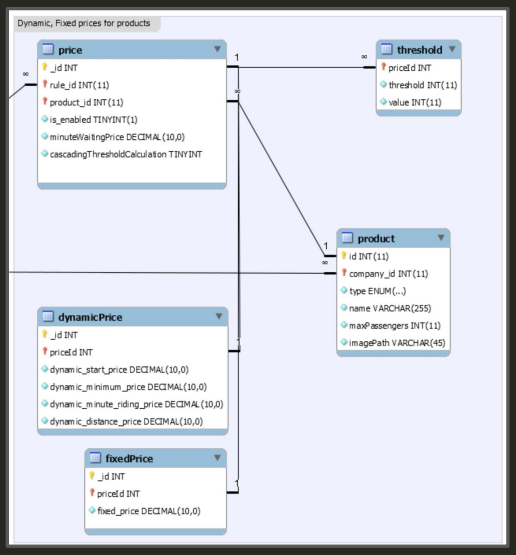


Old:

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
2. Aggregate
    a. Now performs 1 query instead of sum(vehicleTypes) queries

Flow

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)
8. The Price calculator promises a calculation for each pricing rule (per vehicle type)
9. If a promise fails, an empty array is returned
10. 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,
       },
     },
   });
 }
```

End

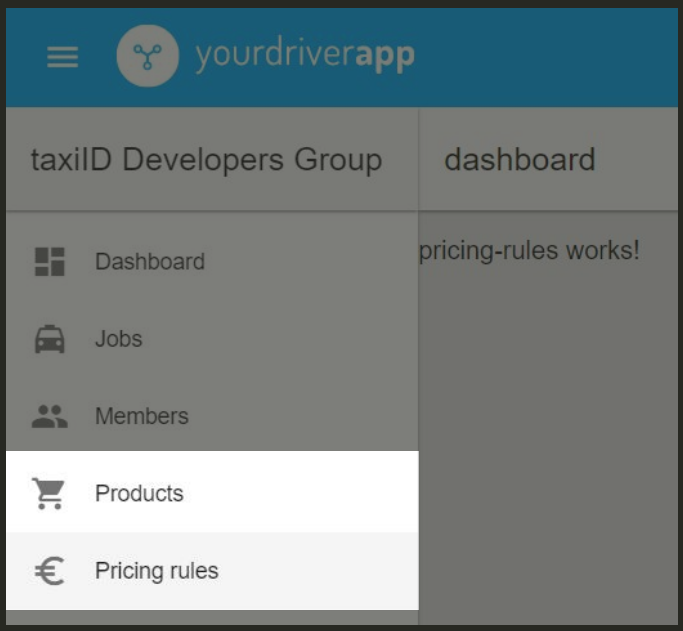
B.5 Sprint 3 - review

Sprint 3
Portal: products & rules

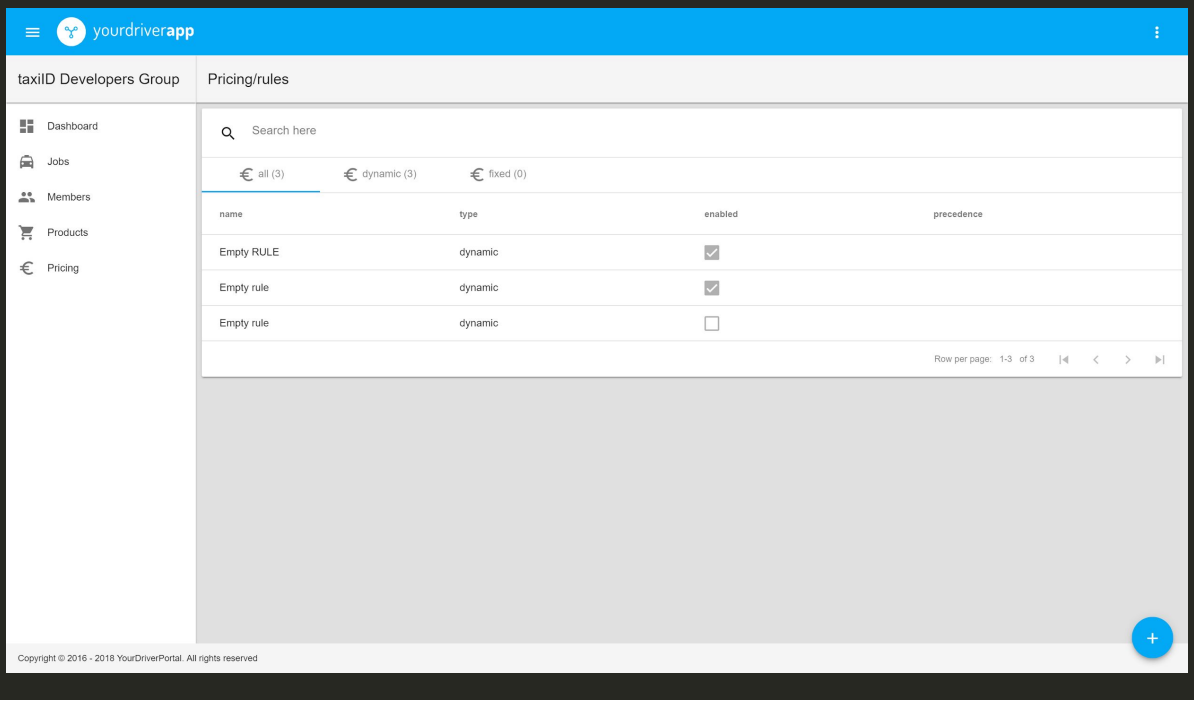
products and pricing rules

Views

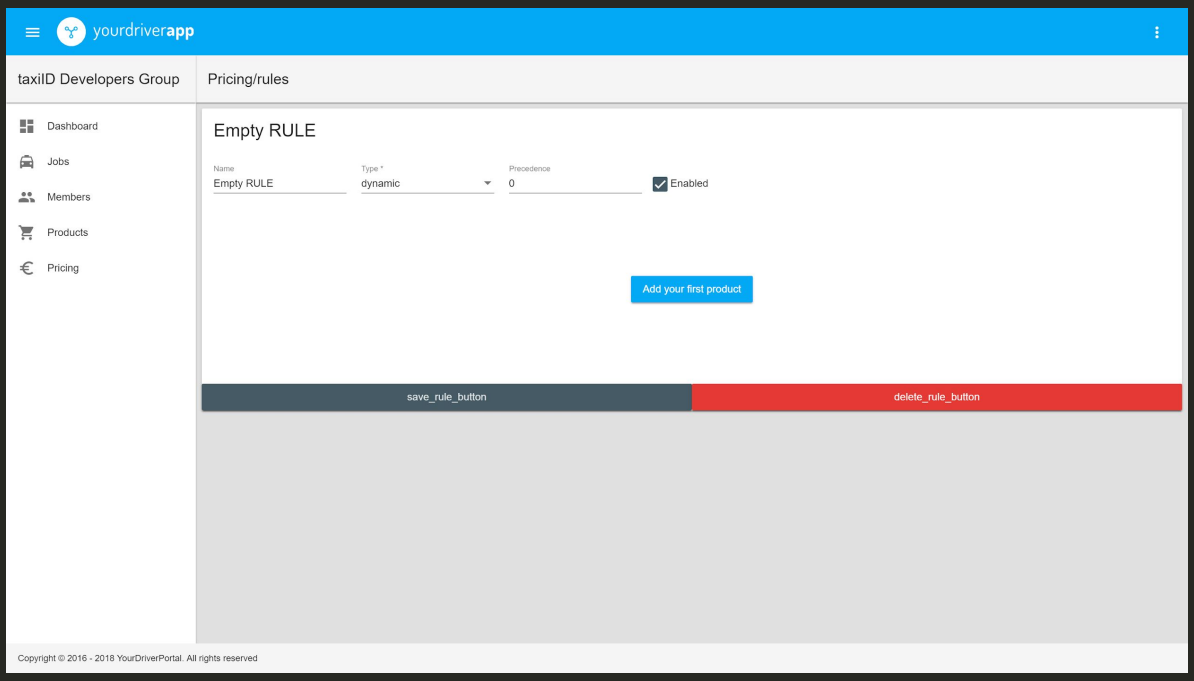
Sidebar



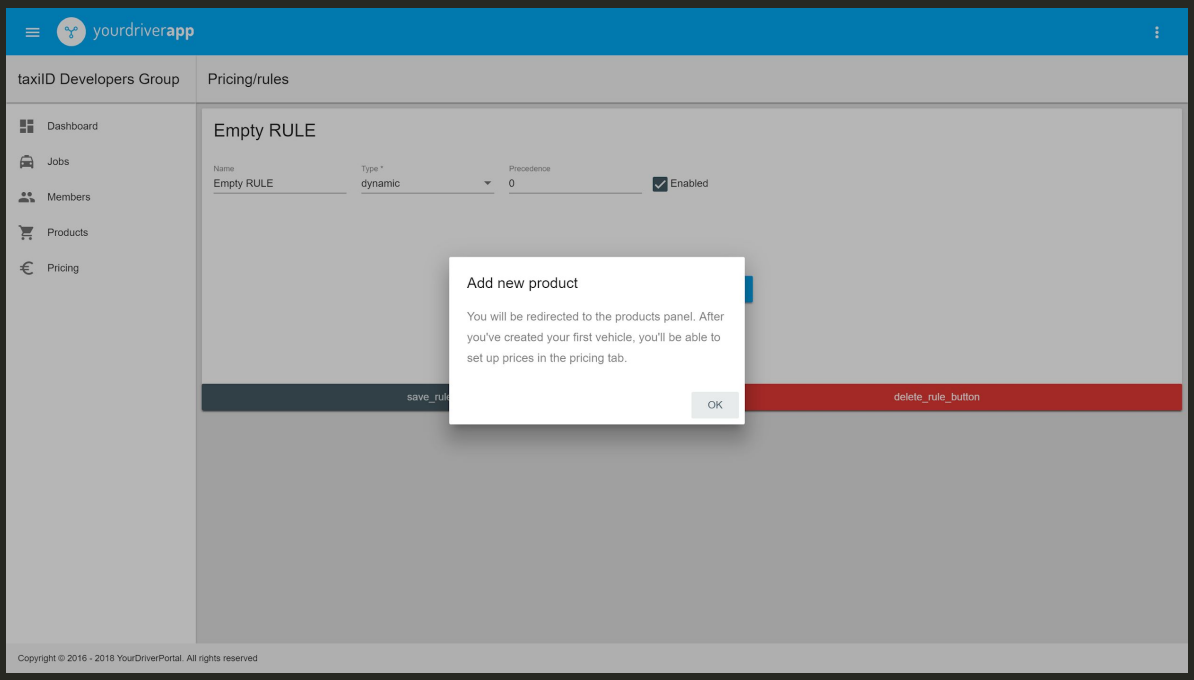
Pricing/rules



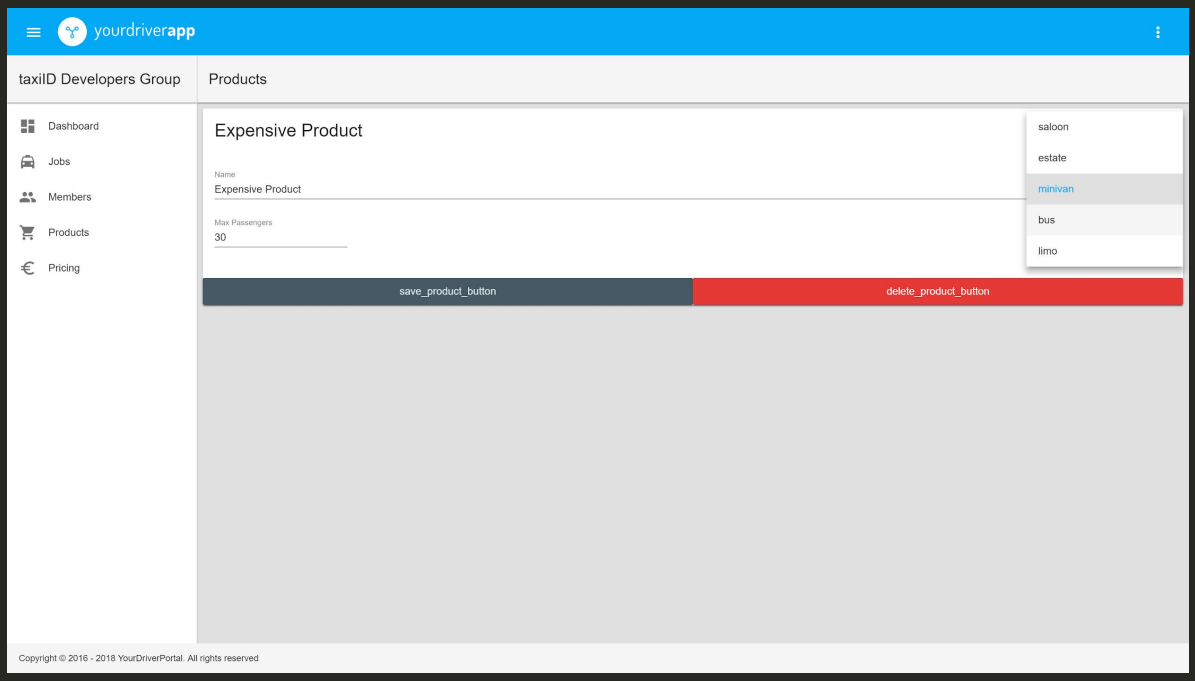
Empty Pricing/rules



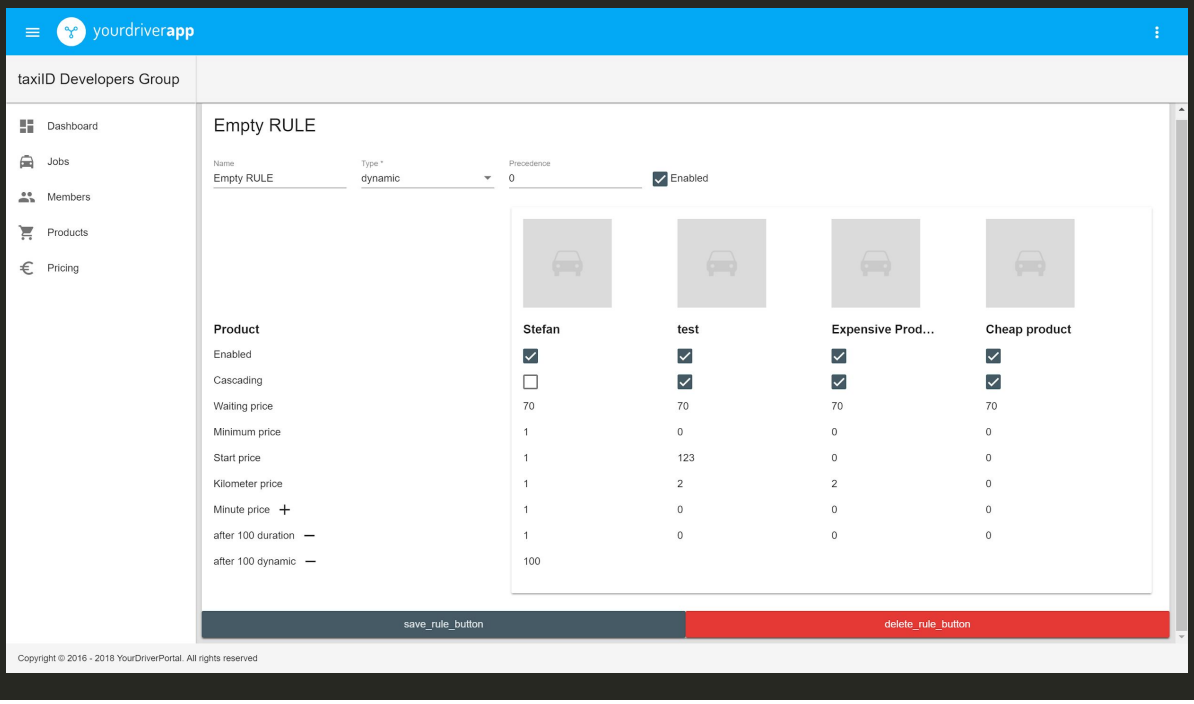
Add first product Pricing/rules



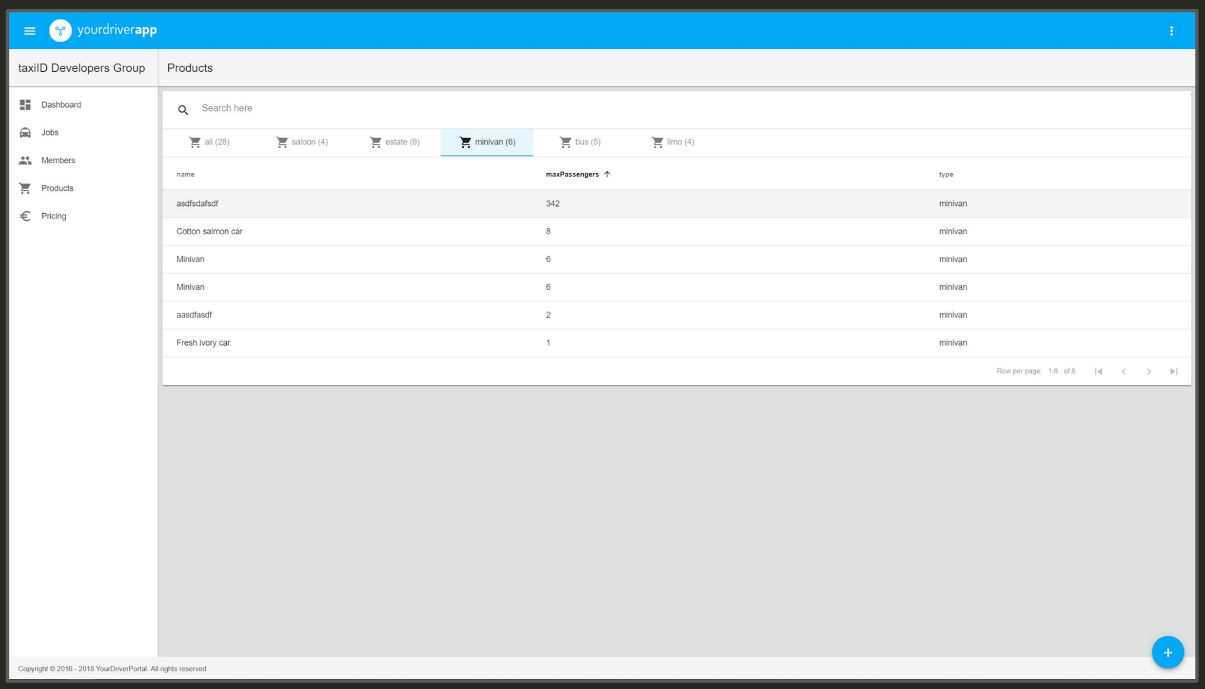
Edit Product



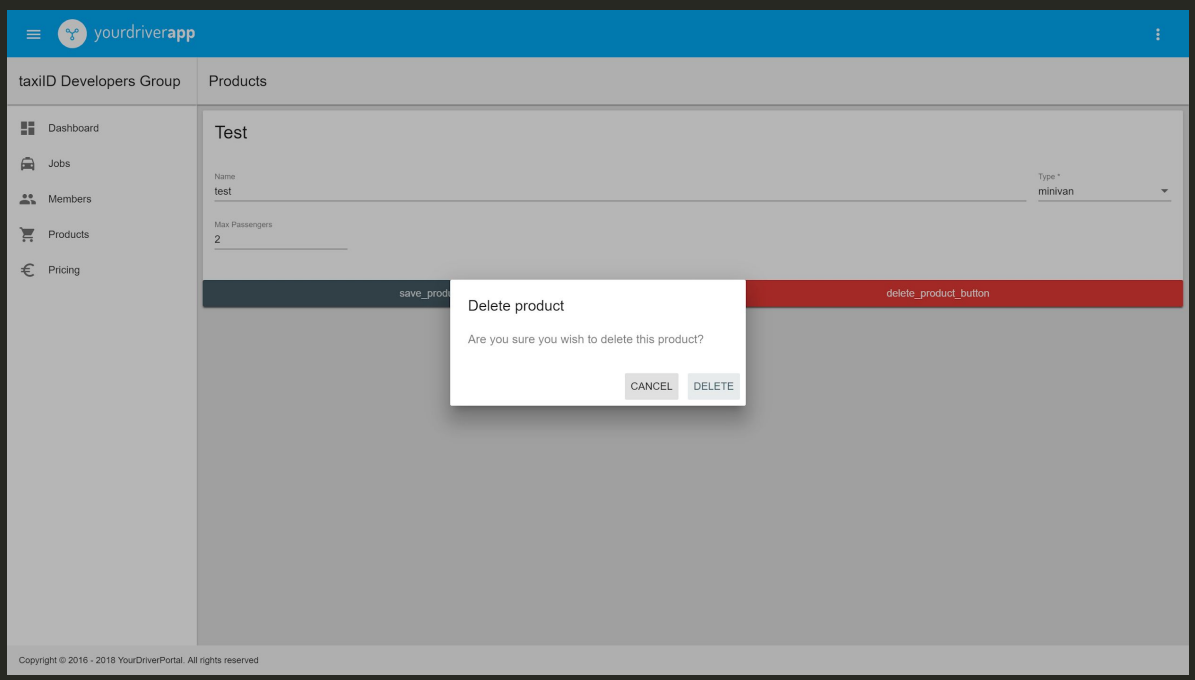
Edit Pricing/rule



Products



Delete Product



Management

Related models are automatically inserted

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.

Todo

Todo 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.

Todo 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.

End

B.6 Sprint 4 - review

Sprint 4
Portal: apps & special rates

sortable, app associated, timeframe restricted pricing & special rates

PM2 / TS compatibility

PM2 / TS Compatibility

Best case scenario: (on server)

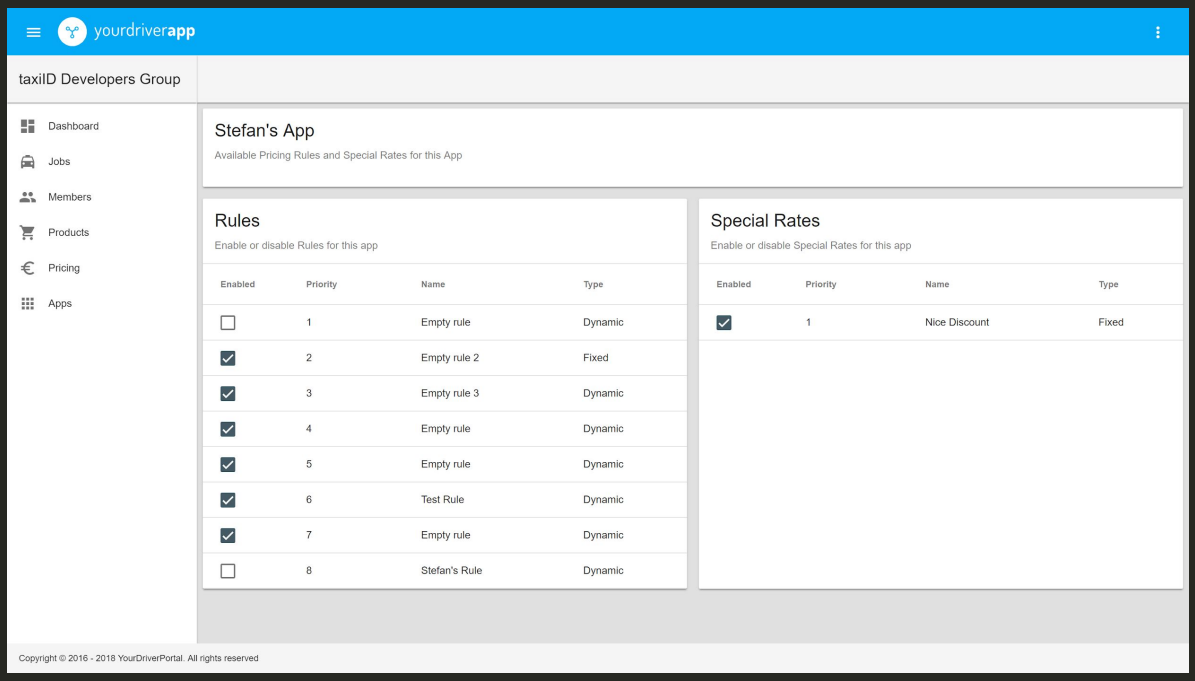
1. `$ npm install pm2@latest -g`
2. `$ pm2 update`
3. `$ pm2 install typescript@2.6.2`

Worst case scenario solution: (locally)

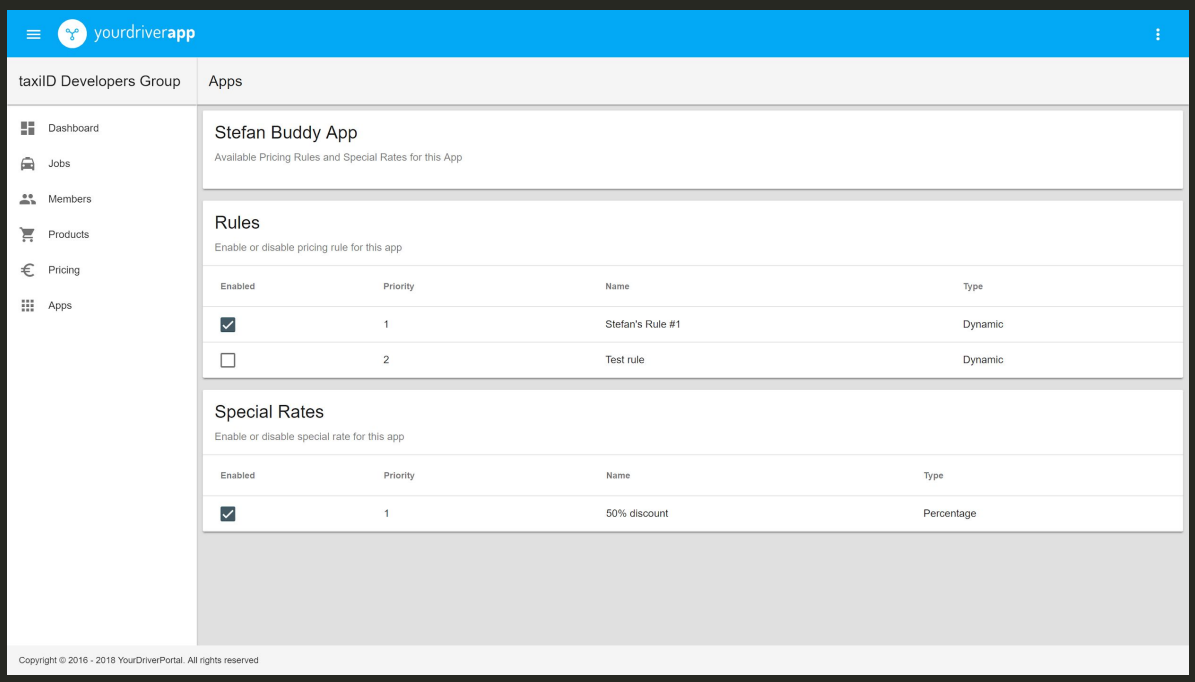
1. Transpile .ts files locally
2. Push resulting .js files to the repository
3. Run project like normal on the server

Views

Apps, Enabled Rules and Special Rates (v1)

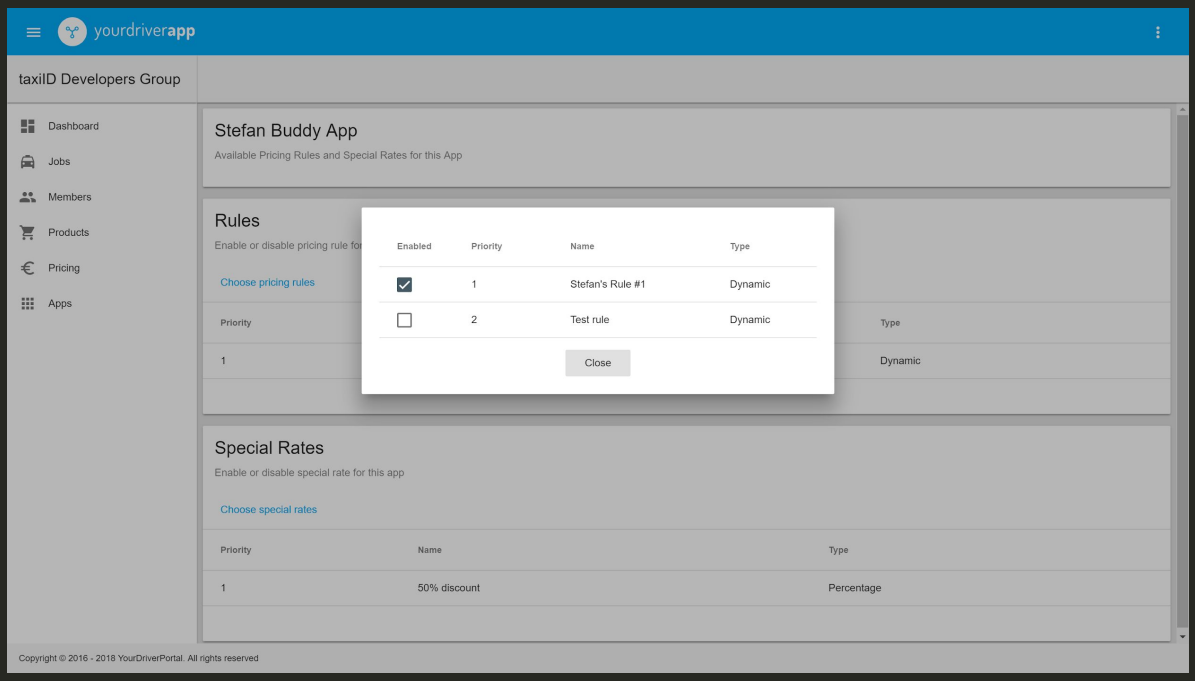


Apps, Enabled Rules and Special Rates (v2)



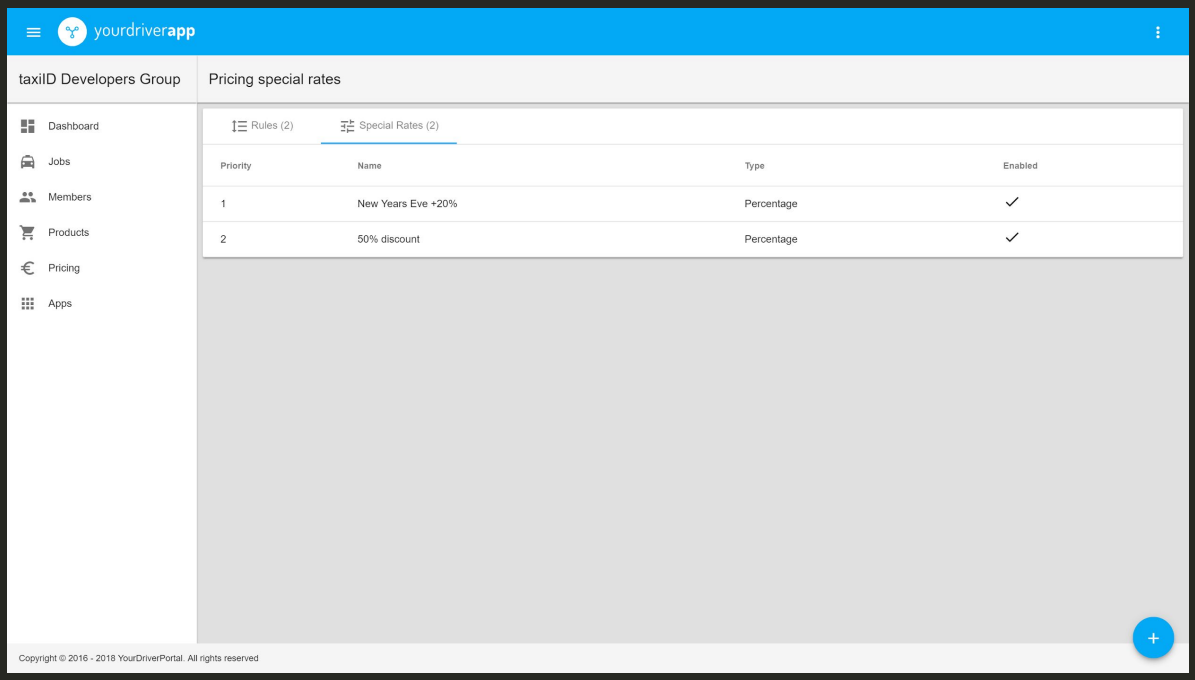
Apps, Enabled Rules and Special Rates (new)

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.



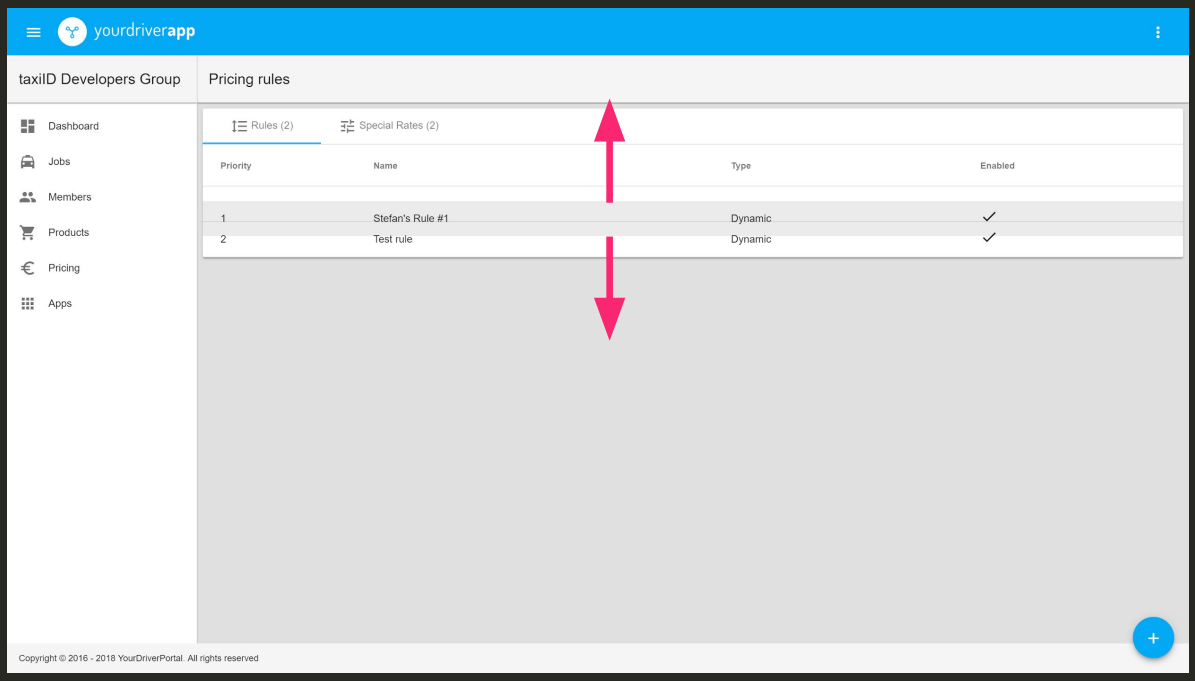
Pricing Rules / Special Rates (new tab)

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



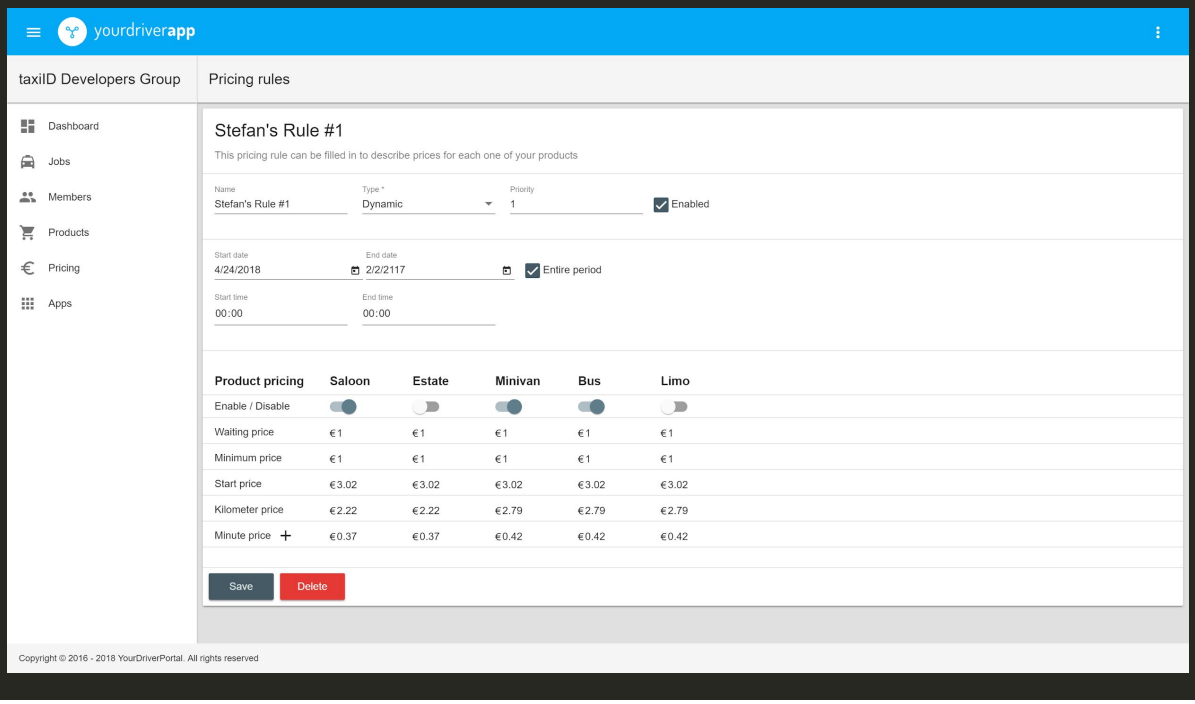
Dragging Pricing Rules / Special Rates

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



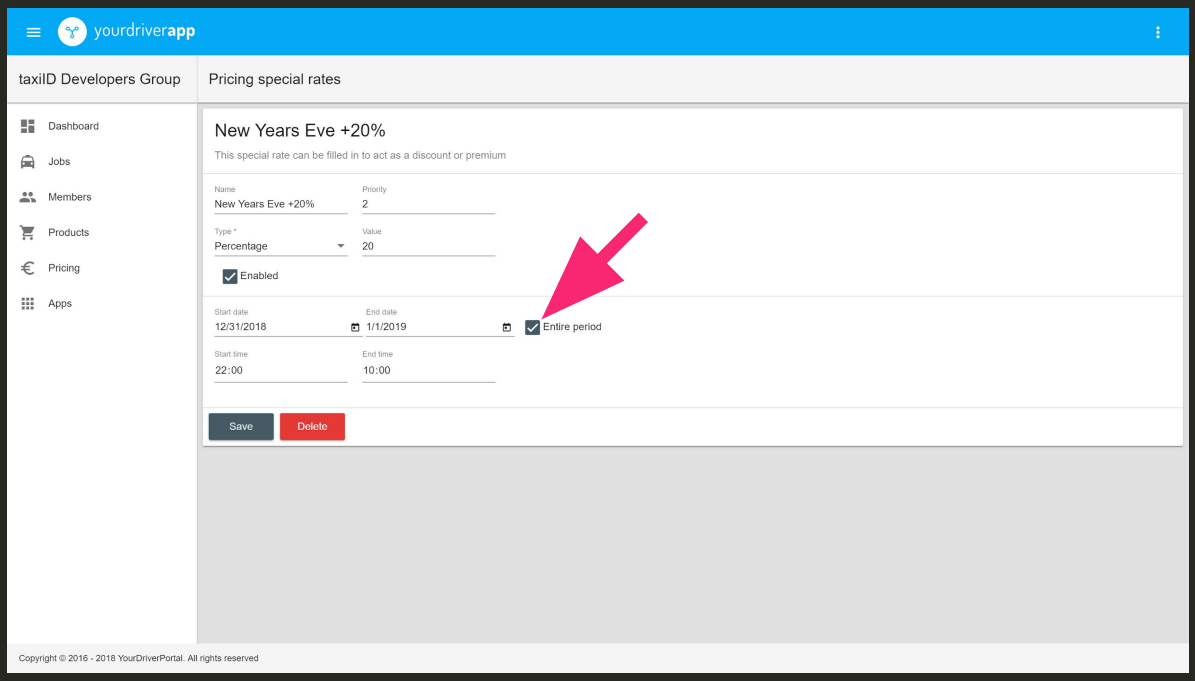
Pricing Rule (added timeframe selector)

A timeframe is added to the pricing rule as a component.



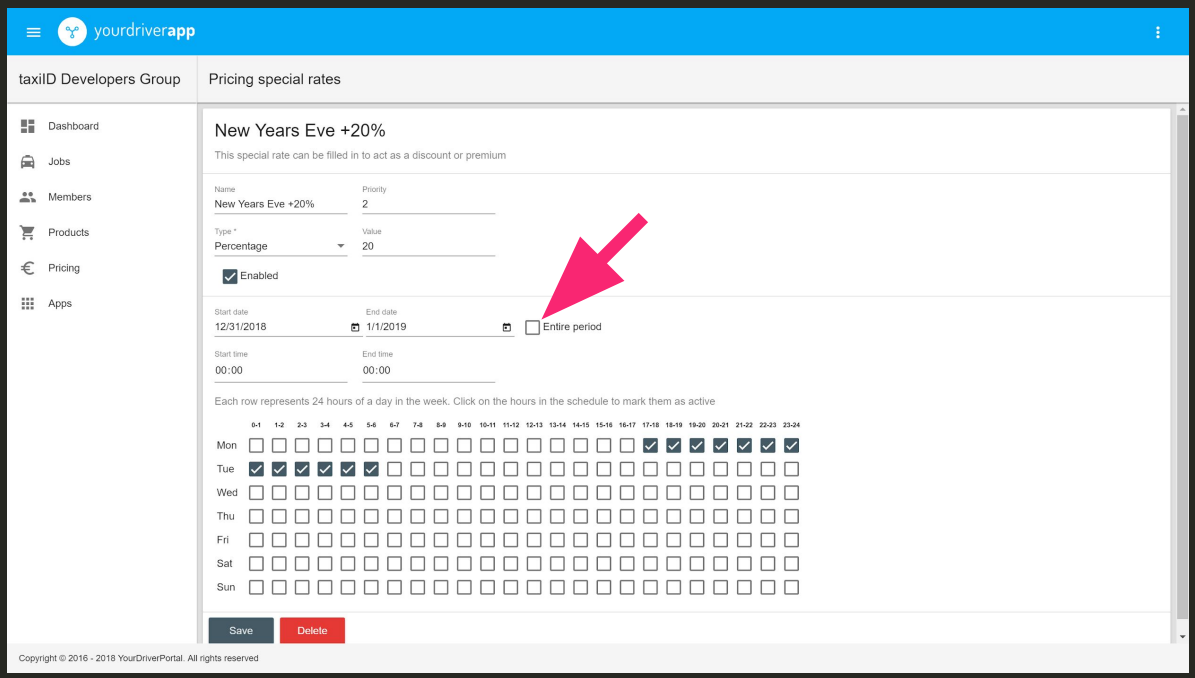
Special Rate (new page)

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



Special Rate (new timeframe hour selector)

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.



Timeframes

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.

```
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
       10100000010111011001000
       11001000000100001010110 1
       01011110100000101001110”

    )
})
```

Inserting Timeframes:

Type 0 BinData is a generic type.

```
// No example
```

Querying Timeframes:

Type 0 BinData is a generic type.

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.

```
db.Timeframe.insert({

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

})
```

Inserting Timeframes:

An array of booleans. Can be quite verbose.

```
db.Discount.find({

  timeframes: {

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

  }

})
```

Querying Timeframes:

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

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 in solving that.

```
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

})
```

Inserting Timeframes:

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

```
// 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/
```

Querying Timeframes:

Management

Related models are automatically mutated

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 default timeframe is added ~~to a new entity, each day starting from 7:00 and ending on 18:00, from now until 100 years in the future~~ by the frontend.

Todo

Todo 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. 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.

Todo 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.

End