A rule-based geospatial reasoning system for trip price calculations

Stefan Schenk

Supervisor: Willem Brouwer

Advisor: Mewis Koeman

Department of Software Engineering
Amsterdam University of Applied Sciences

This dissertation is submitted for the degree of
Bachelor Software Engineering

April 2018

Todo list

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 for taxi companies. Hailing a taxi has rarely been performed by sticking out ones hand, hoping to catch the attention of a bypassing cab ever since. Recently, taxiID has started developing a new brand called YourDriverApp (YDA), a lighter and newer version of the original solution, being more focussed on smaller taxi companies. Despite the fact that YDA is new, it still depends on the price calculation functionality of the legacy system. This chapter expands on this matter and how it was translated into an assignment.

1.1 Context

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

1.2 Problem Definition

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

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

Fig. 1.1 Comma Separated File containing Fixed Prices in cents

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

kks32: Refer to thresholds

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

1.3 Assignment

The title of this thesis reads:

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

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

1.4 Research

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

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

1.4.1 Questions

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

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

that could be used in every country?

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

1. In what way can locations be represented to be universally interpretable?
    1.1. Which types of locations should be distinguished?
    1.2. What are the main differences between postal systems used around the globe?
    1.3. Can postal codes be abstracted to geospatial data while retaining the same usefulness in the system?

1.4. Which Database Management Systems (DBMS) cover the location storage use cases for this project?

2. What is most fitting solution to integrate TPS and UI into the existing architecture?

   2.1. Which architectural patterns fit in with the exising architecture?

   2.2. Which DBMS is suited for this project?

   2.3. How can authentication between services be implemented or improved?

3. TPS

   3.1. a

4. How can complex pricing rules be communicated through the UI?

   4.1. Which views are essential?

   4.2. How should locations be defined and managed by the user?

   4.3. How should timeframes be represented in the interface?

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

1.5 Process

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

Chapter 2

Encoding Locations

2.1 Introduction

To answer in what way locations can be represented to be universally interpretable, the definition of a location must be well understood. Encoding of locations has historically been of great importance, and is always being modernized. This chapter aims to find the best method of representing locations that is universally interpretable and most importantly usable in this project. Locations, plurally, suggesting that more than one location should be encoded. This is where the challenge lies.

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.

kks32: galileo

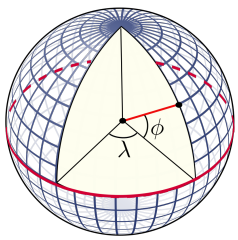


Fig. 2.1 A perspective view of the Earth showing how latitude and longitude are defined on a spherical model.

Modern navigation relies on sattelites that are capable of providing information to determine a location with an accuracy of 9 meters. The precision of hybrid methods using cell towers and Wi-Fi Location Services enables precise tracking of modern devices. Latitude and longitude are often referred to as GPS coordinates, as GPS is often used to calculate latitude and longitude by receiving position and time code sequences from at least four sattelites. Addresses are another representation of a location used in navigation. Addresses are easier to communicate than a pair of GPS coordinates, but can be ambiguous, imprecise, inconsistent in format. Addresses commonly make use of Postal Code systems, which have reliably been assigned to geographical areas with the purpose of sorting mail. Although even today, there are countries that do not have a Postal Code system. 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 before mentioned of this project.

2.3 Useful Location Types

While setting up a backlog, a shared knowledge about the terminology used in the issues must be achieved. In the pregame document, the term "area" was defined as a collection of three or more coordinate pairs, or a collection of postal codes. A "point" was defined as one distinct coordinate pair or one distinct postal code. This way a location could either be an area or a point, with which all possibilities are covered. As stated in Appendix A, the definition of an area is precise, unambiguous and easy to use in compare in computer programs. A single point may match another single point if it's the exact same point. A point may be sitting on top of a line or is contained within an area. The only other option is the negation of these statements. Because use cases for lines will be non-existent, points and areas are the proper candidates for spatial queries.

2.4 Requisites of Location types

A taxi company director wants to be able to set price or define discounts from or to a certain location. They would like to define prices based only on departure locations, or only on destination locations, or both. For example: 'to Schiphol, a trip should cost €10,-', or 'from van der Valk hotels, a trip should cost €5,-', or 'from van der Valk hotels to Schiphol, the km price should be €0,60'. In the current implementation, a record would be stored containing

departure location, destination location and price for every combination, where locations were defined as zip codes. Instead, it would make sense to be able to reuse locations after they have been defined once.

2.5 Literature Review

2.6 Database Prerequisites

The database must be capable of determining whether a virtual perimeter contains a set of coordinates, more specifically, it must adhere to The Open Geospatial Consortium (OGC) Simple Feature Access ISO 19125-1 [2] and ISO 19125-2 [3], including spatial data types, analysis functions, measurements and predicates for this requirement, or have some comparable implementation. The scenario presented in image 2.2 should be replicable.

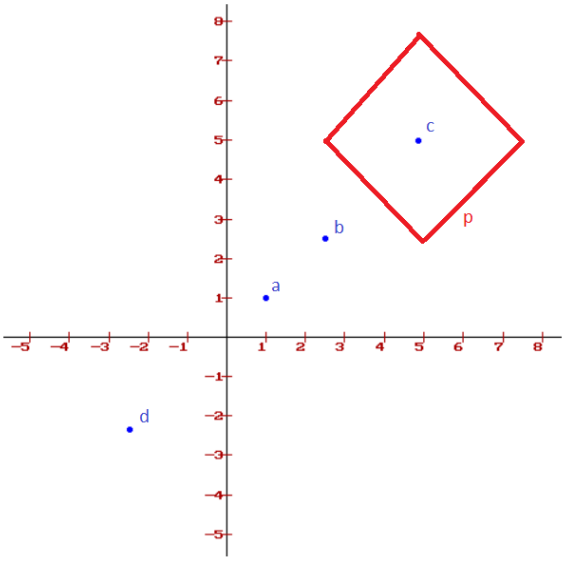


Fig. 2.2 Four Points, one Polygon p containing Point c.

2.6.1 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 [4] and MYSQL documentation [5] could be consulted. When a generic geometry column, or point column is created, points can be inserted as shown in snippet 2.1 and 2.2

```
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;
```

Listing 2.1 Insert four points

```
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;
```

Listing 2.2 Insert polygon

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

kks32: Add ref to snippet

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

kks32: Make multicol

```
1  // All points contained in polygon
2  SELECT ST_ASTEXT(POINT)
3  FROM POINT
4  WHERE
5  ST_CONTAINS(
6       (
7         SELECT POLYGON
8         FROM POLYGON
9         WHERE id = 1
10      ),
11      POINT
12    )
13
14 // All polygons containing point
15 SELECT ST_ASTEXT(POLYGON)
16 FROM POLYGON, POINT
17 WHERE
18   POINT.id = 3 AND ST_CONTAINS(
19     POLYGON.polygon,
20     POINT.point
21   )
```

2.6.2 OpenGIS Incompatible databases

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

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

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

kks32: Add ref to image

is recreated in MongoDB using queries shown in snippet

kks32: Add ref to snippet

.

```
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' })
```

kks32: Make multicol

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

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

2.7 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

kks32: Add reference to Agarwal and Rajan

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

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

after the cited paper Schmid et al. 2015

kks32: Add ref to Stephan Schmid Eszter Galicz

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

In what way can locations be represented to be universally interpretable?

1. Which types of locations should be distinguished?
2. What are the main differences between postal systems used around the globe?
3. Can postal codes be abstracted to geospatial data while retaining the same usefulness in the system?
4. How can different types of locations be effectively stored in a database?

Chapter 3

System Architecture

3.1 Introduction

The price calculation system that is being made should integrate in the existing architecture seamlessly. This chapter aims to answer question number two, and its subquestions. First it is determined whether the frontend and backend that are to be developed should be integrated in existing projects, or should be made in separate projects altogether. If the backend is created as a separate project, authentication and authorization are directly affected by these decisions. Separation implies more complex identity management, or less separation of concern. Then, the database should be capable of storing geometry, and accept queries to determine which polygons contain a set of points, and which points are contained within a polygon.

3.2 Architectural Patterns

The current system architecture consists of three public API's and eight private API's that connect to four databases, as can be seen in 3.1 The core API is the interface to which mobile applications make requests. It is possible to integrate the pricing system as a module in the existing project. This simplifies authentication and authorization, because it already exists is that project. Another option is to implement the system as a microservice, which is infamously known as a service-oriented architecture (SOA). A microservice architecture is an architectural style that focuses on loosely-coupled services, improving scaling and continuous development of complex applications. Each microservice is responsible for managing and containing state to enable users who would like to use the system to be authenticated and authorized. Advantages of a microservice are the fact that a microservice is a self-contained and naturally modular structure, but authentication and authorization must be handled by the

microservice itself, unless state is shared amongst services, which would eliminate the reason to use a microservice at all. In the present architecture, different services implement different authentication methods, store different information about different users. Authorization is managed by sending extra headers for each crucial piece of information. For example: company information, application information and user information are all sent in separate headers.

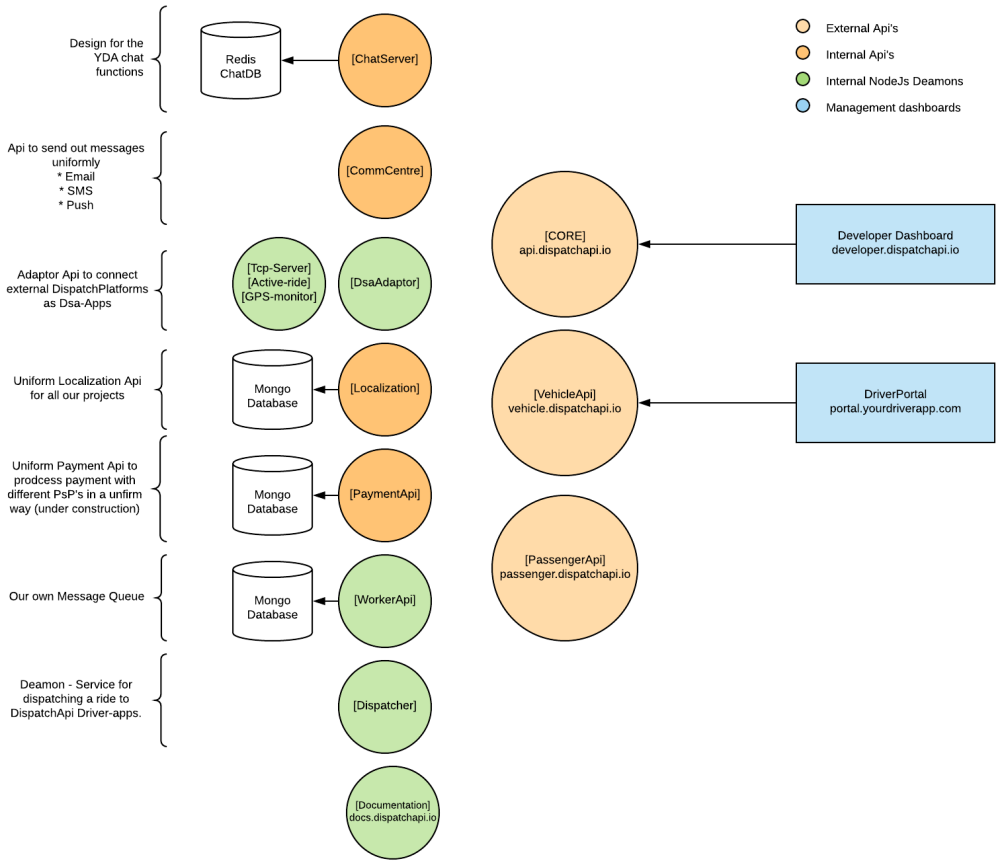


Fig. 3.1 Current System Architecture

When the amount of services that are added to the architecture increases, the mount of information that is no longer centralized increases as well.

3.3 Authentication and Authorization

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

- Example 1: The microservice authenticates and authorizes users all by itself, managing sessions and storing user data in its database.
- Example 2: The microservice connects to an 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.

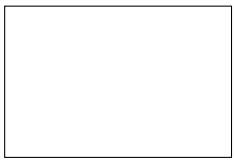
In the first example, the microservice seems to work independently, because it has knowledge about the users identity without making requests to adjecent systems, or connecting to external databases. But this is not true. If data about the user is mutated in the core system, the microservice needs to 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.

3.3.1 JSON Web Tokens

Example three entirely removes the database connection to any user data. This is possible when a JSON Web Token (JWT) is used. A JWT may be signed with a cryptographic

algorithm or even a public/private key pair using RSA. After the user enters valid credentials, the core system validates the credentials by comparing them with user data in the database. The core system signs a token that with a secret that is known by the microservice. The token consists of three parts, separated by a fullstop. The first part (header) of the token contains information about the hashing algorithm that was used to encrypt the payload. This part is Base64Url encoded. The payload itself contains information stored in JSON format:

kks32: Show diagram with hierarchy of companies and apps



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

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

3.3.2 oAuth 2.0

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

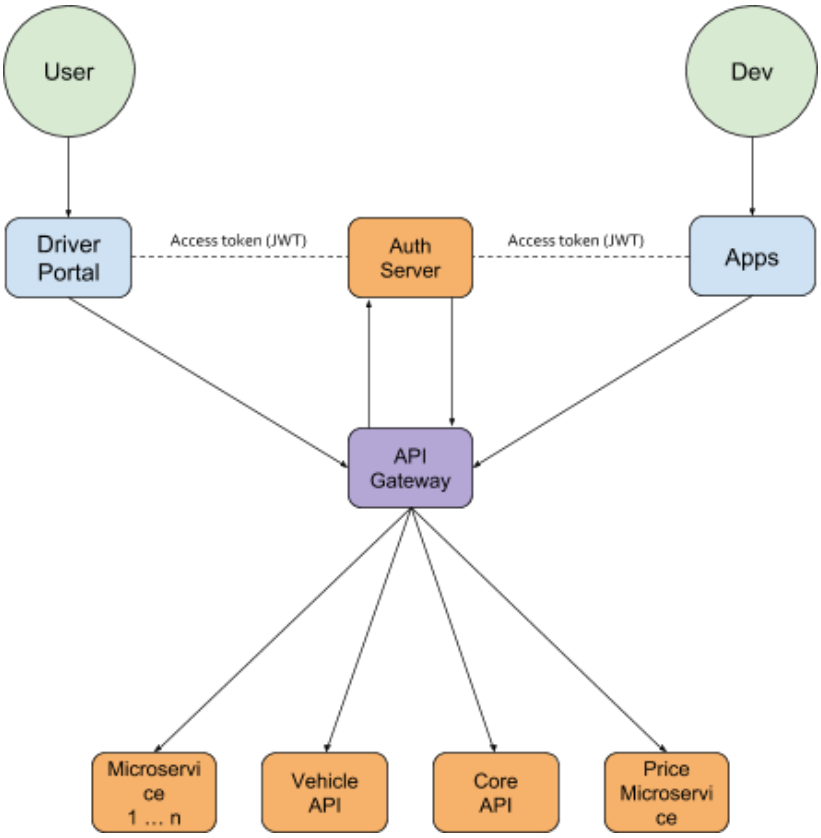


Fig. 3.2 oAuth, API Gateaway, using JWT

3.3.3 API Gateway

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

Next to authentication, the gateway could optimize the endpoints so that no multiple requests are needed from external agents to gather different types of resources. These calls could be made internally to the microservices behind the gateway. This also opens the possibility 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.

Chapter 4

Realization

4.1 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.

4.2 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.

4.3 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.

4.4 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 hold the identifiers in the payload, so that the TPS can use the identifiers after decrypting the token. Companies have one country 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.

4.5 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. This is automatically maintained whenever a product is added or deleted. Furthermore, threshold rules can be added or deleted for distances and durations, making this particular view very complex. This final task was not finished in time.

4.6 Sprint 4 - Expanding the Portal

- thresholds continued - processing feedback - pricing rules overview, rules must be draggable to prioritize - priority must be maintained distinctly - app installations must be displayed - pricing rules and special rates must be enabled for specific app installs - allow on meter calculations - internationalization - timeframes -

4.7 Result

Chapter 5

Proposed Portal Solution

5.1 Introduction

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

5.2 Required Views

5.3 Methods and Techniques

5.4 Proposal Pricing Rules View

5.5

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

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

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

Chapter 6

Conclusion

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

Chapter 7

Conclusion

Chapter 8

Recommendations

References

[1] U. T. Inc., “The uber story,” 2018.

[2] “Simple feature access - part 1: Common architecture | ogc,” 2018.

[3] “Simple feature access - part 2: Sql option | ogc,” 2018.

[4] “Postgis 2.4.5dev manual,” 2018.

[5] “Mysql 5.7 reference manual - geometry class,” 2018.

List of figures

List of tables

Appendix A

Pregame

Pregame

TeXLive package - full version

1. Download the TeXLive ISO (2.2GB) from
   https://www.tug.org/texlive/
2. Download WinCDEmu (if you don't have a virtual drive) from
   http://wincdemu.sysprogs.org/download/
3. To install Windows CD Emulator follow the instructions at
   http://wincdemu.sysprogs.org/tutorials/install/
4. Right click the iso and mount it using the WinCDEmu as shown in
   http://wincdemu.sysprogs.org/tutorials/mount/
5. Open your virtual drive and run setup.pl

or

Basic MikTeX - TEX distribution

1. Download Basic-MiKTEX(32bit or 64bit) from
   http://miktex.org/download
2. Run the installer
3. To add a new package go to Start » All Programs » MikTex » Maintenance (Admin)
   and choose Package Manager

4. Select or search for packages to install

TexStudio - TEX editor

1. Download TexStudio from
   http://texstudio.sourceforge.net/#downloads

2. Run the installer

Mac OS X

MacTeX - TEX distribution

1. Download the file from
   https://www.tug.org/mactex/

2. Extract and double click to run the installer. It does the entire configuration, sit back and relax.

TexStudio - TEX editor

1. Download TexStudio from
   http://texstudio.sourceforge.net/#downloads

2. Extract and Start

Unix/Linux

TeXLive - TEX distribution

Getting the distribution:

1. TexLive can be downloaded from
   http://www.tug.org/texlive/acquire-netinstall.html.

2. TexLive is provided by most operating system you can use (rpm,apt-get or yum) to get TexLive distributions

Installation

1. Mount the ISO file in the mnt directory

   ```
   mount -t iso9660 -o ro,loop,noauto /your/texlive####.iso /mnt
   ```

2. Install wget on your OS (use rpm, apt-get or yum install)

3. Run the installer script install-tl.

   ```
   cd /your/download/directory
   ./install-tl
   ```

4. Enter command 'i' for installation

5. Post-Installation configuration:
   http://www.tug.org/texlive/doc/texlive-en/texlive-en.html#x1-320003.4.1

6. Set the path for the directory of TexLive binaries in your .bashrc file

For 32bit OS

For Bourne-compatible shells such as bash, and using Intel x86 GNU/Linux and a default directory setup as an example, the file to edit might be

```
edit $~/.bashrc file and add following lines
PATH=/usr/local/texlive/2011/bin/i386-linux:$PATH;
export PATH
MANPATH=/usr/local/texlive/2011/texmf/doc/man:$MANPATH;
export MANPATH
INFOPATH=/usr/local/texlive/2011/texmf/doc/info:$INFOPATH;
export INFOPATH
```

For 64bit OS

```
edit $~/.bashrc file and add following lines
PATH=/usr/local/texlive/2011/bin/x86_64-linux:$PATH;
export PATH
MANPATH=/usr/local/texlive/2011/texmf/doc/man:$MANPATH;
export MANPATH
```

```
INFOPATH=/usr/local/texlive/2011/texmf/doc/info:$INFOPATH;
export INFOPATH
```

Fedora/RedHat/CentOS:

```
sudo yum install texlive
sudo yum install psutils
```

SUSE:

```
sudo zypper install texlive
```

Debian/Ubuntu:

```
sudo apt-get install texlive texlive-latex-extra
sudo apt-get install psutils
```

Appendix B

Installing the CUED class file

LaTeX.cls files can be accessed system-wide when they are placed in the <texmf>/tex/latex directory, where <texmf> is the root directory of the user's TeXinstallation. On systems that have a local texmf tree (<texmflocal>), which may be named "texmf-local" or "localtexmf", it may be advisable to install packages in <texmflocal>, rather than <texmf> as the contents of the former, unlike that of the latter, are preserved after the LaTeXsystem is reinstalled and/or upgraded.

It is recommended that the user create a subdirectory <texmf>/tex/latex/CUED for all CUED related LaTeXclass and package files. On some LaTeXsystems, the directory look-up tables will need to be refreshed after making additions or deletions to the system files. For TeXLive systems this is accomplished via executing "texhash" as root. MIKTeXusers can run "initexmf -u" to accomplish the same thing.

Users not willing or able to install the files system-wide can install them in their personal directories, but will then have to provide the path (full or relative) in addition to the filename when referring to them in LaTeX.