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

🟩 What are the main differences between postal systems used around the globe? . . 8
🟩 talk about use cases and usefulness of geospatial data 9
🟩 See if other people solved locations 10
🟩 Add ref to snippet 11
🟩 Add ref to Geospatial Query Operators — MongoDB Manual 3.6 12
🟩 Add ref to image 12
🟩 Add ref to snippet 12
🟩 Add reference to Agarwal and Rajan 13
🟩 Add reference to Geospatial Performance Improvements in MongoDB 3.2," MongoDB 13
🟩 Add ref to Stephan Schmid Eszter Galicz 13
🟩 Show diagram with hierarchy of companies and apps 21
🟩 Write chapter about methods and technologies 25
🟩 Why? 27
🟩 This is not researched yet, as it's covered in later sprints 29

Table of contents

1 Introduction **1**

1.1 Context 1

1.2 Problem Definition 2

1.3 Assignment 3

1.4 Research 3

1.4.1 Questions 4

1.5 Process 5

2 Encoding Locations **7**

2.1 Introduction 7

2.2 A Brief History Of Geographic Locations 7

2.3 Requisites of Location Types 8

2.3.1 Location Related Scenarios 9

2.4 Literature Review 10

2.5 Database Prerequisites 10

2.5.1 OpenGIS Compatible databases 10

2.5.2 OpenGIS Incompatible databases 12

2.6 Performance and Clustering Trade-offs 13

3 System Architecture **15**

3.1 Introduction 15

3.2 Architectural Patterns 15

3.2.1 Monoliths 15

3.2.2 Microservices 17

3.2.3 Frontend and Backend 17

3.3 Information Dependencies 17

3.4 Authentication and Authorization 19

3.4.1 The User 20

3.4.2 Statelessness 20
3.4.3 JSON Web Tokens 21
3.4.4 oAuth 2.0 21
3.4.5 API Gateway 25
3.5 Suitability of Methods and Techniques 25
3.6 Conclusion 25

4 Trip Price Calculation System **25**
4.1 Introduction 25
4.2 Breakdown 25
4.3 Timeframes 27
4.3.1 Conventional Approach 27
4.3.2 Bitmap 28
4.4 Data Model 28
4.5 Logical Flow of a Price Calculation 28

5 Proposed Portal Solution **29**
5.1 Introduction 29
5.2 Required Views 29
5.3 Methods and Techniques 29
5.4 Proposal Pricing Rules View 29
5.5 29

6 Realization **31**
6.1 Introduction 31
6.2 Methods and Techniques 31
6.3 Sprint 1 - Dynamic Price Calculations 31
6.4 Sprint 2 - Authentication and Authorization 32
6.5 Sprint 3 - Setting up the Portal 32
6.6 Sprint 4 - Expanding the Portal 32
6.7 Sprint 5 - Expanding the Portal 33
6.8 Result 33

7 Conclusion **35**

8 Recommendations **37**

References **39**

List of figures **41**

List of tables **43**

Appendix A Pregame **45**

Appendix B Sprint Review and Proposal Slides **81**

B.1 Sprint 1 - review 81
B.2 Sprint 2 - breakdown 93
B.3 Sprint 2 - authentication 105
B.4 Sprint 2 - review 114
B.5 Sprint 3 - review 130
B.6 Sprint 4 - review 139

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 [7].

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 [8]. 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

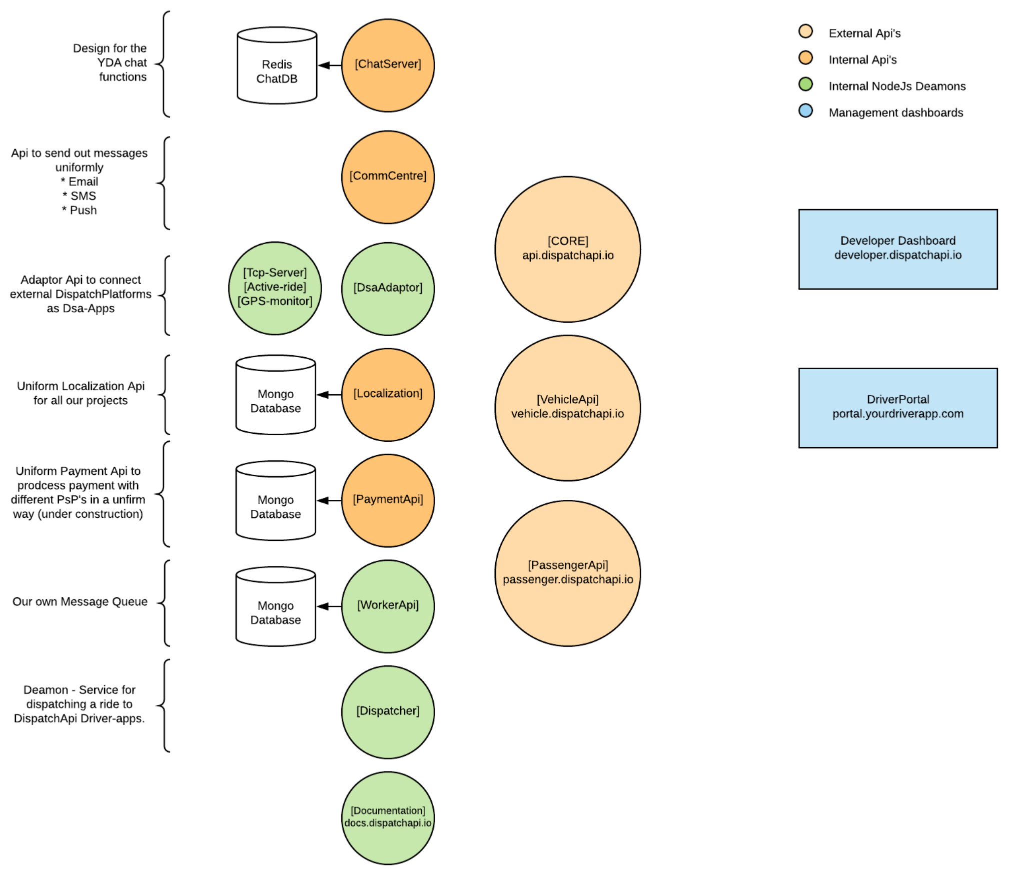
Design for the YDA chat functions
Redis ChatDB
[ChatServer]
Api to send out messages uniformly * Email * SMS * Push
[CommCentre]
Adaptor Api to connect external DispatchPlatforms as Dsa-Apps
[Tcp-Server] [Active-ride] [GPS-monitor]
[DsaAdaptor]
Uniform Localization Api for all our projects
Mongo Database
[Localization]
Uniform Payment Api to prodcess payment with different PsP's in a unfirm way (under construction)
Mongo Database
[PaymentApi]
Our own Message Queue
Mongo Database
[WorkerApi]
Deamon - Service for dispatching a ride to DispatchApi Driver-apps.
[Dispatcher]
[Documentation] docs.dispatchapi.io
[CORE] api.dispatchapi.io
[VehicleApi] vehicle.dispatchapi.io
[PassengerApi] passenger.dispatchapi.io
External Api's
Internal Api's
Internal NodeJs Deamons
Management dashboards
Developer Dashboard developer.dispatchapi.io
DriverPortal portal.yourdriverapp.com

Fig. 3.1 Current System Architecture provided by taxiID

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 [9]. 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, which contradicts an important principle of software engineering; don't repeat yourself (DRY), or dependencies between larger systems, limiting 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 [10], 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 [11], 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 to be operational.

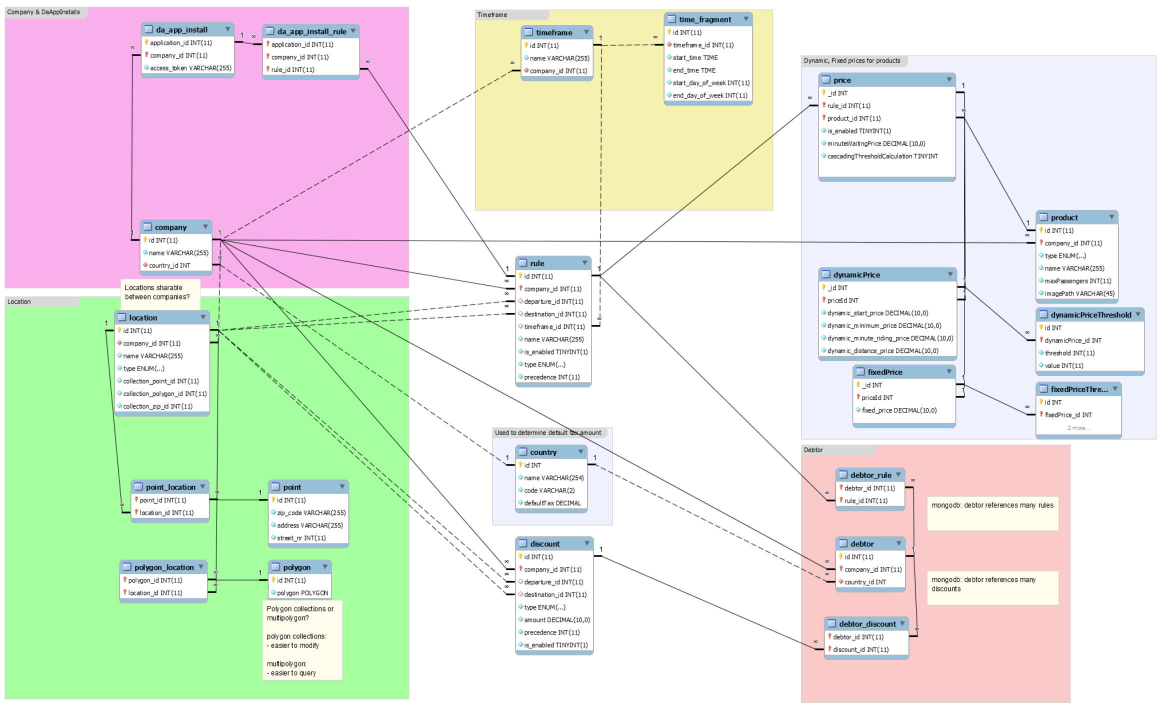

Fig. 3.2 Proposed Database Schema Design

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. 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. 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 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 The User

In the legacy system architecture, different services implement different authentication methods, store different pieces of information of different users. Authorization is managed by sending extra headers for each crucial piece of information, this is clarified in Appendix A, chapter 3.4. For example: company information, application information, and user information is all sent in separate headers. When the amount of services that are added to the architecture increases, the mount of information that is no longer centralized increases with it.

3.4.2 Statelesness

Each microservice is responsible for managing and containing state to enable users who would like to use the system to be authenticated and authorized. Advantages of a microservice are the fact that a microservice is a self-contained and has a naturally modular structure. But authentication and authorization must be handled by the microservice itself, unless state is shared amongst services.

3.4.3 JSON Web Tokens

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

kks32: Show diagram with hierarchy of companies and apps

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

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

3.4.4 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.

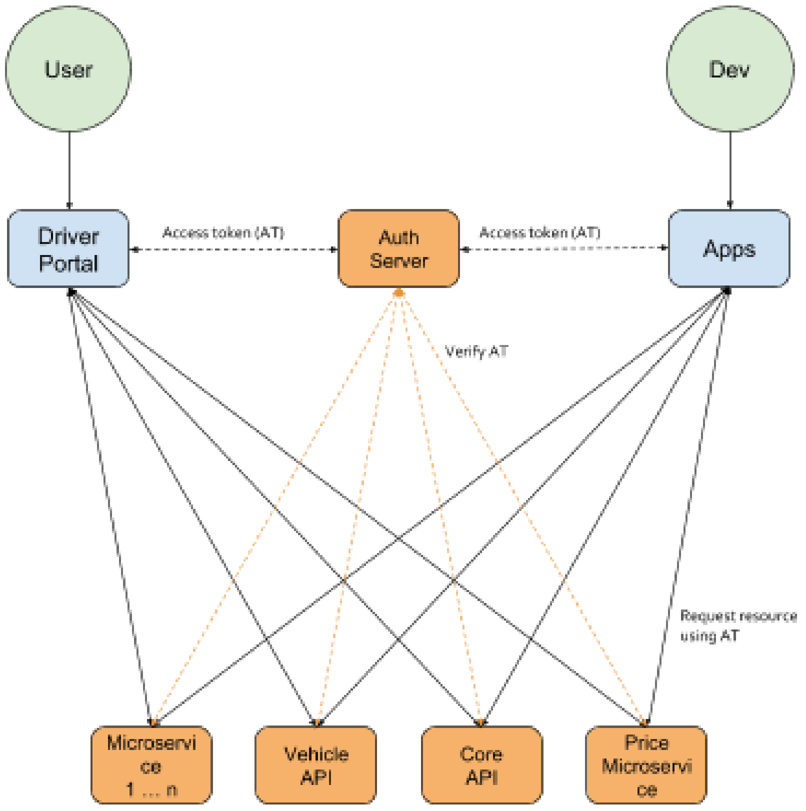

Fig. 3.3 OAuth requests where tokens are verified by Auth Server

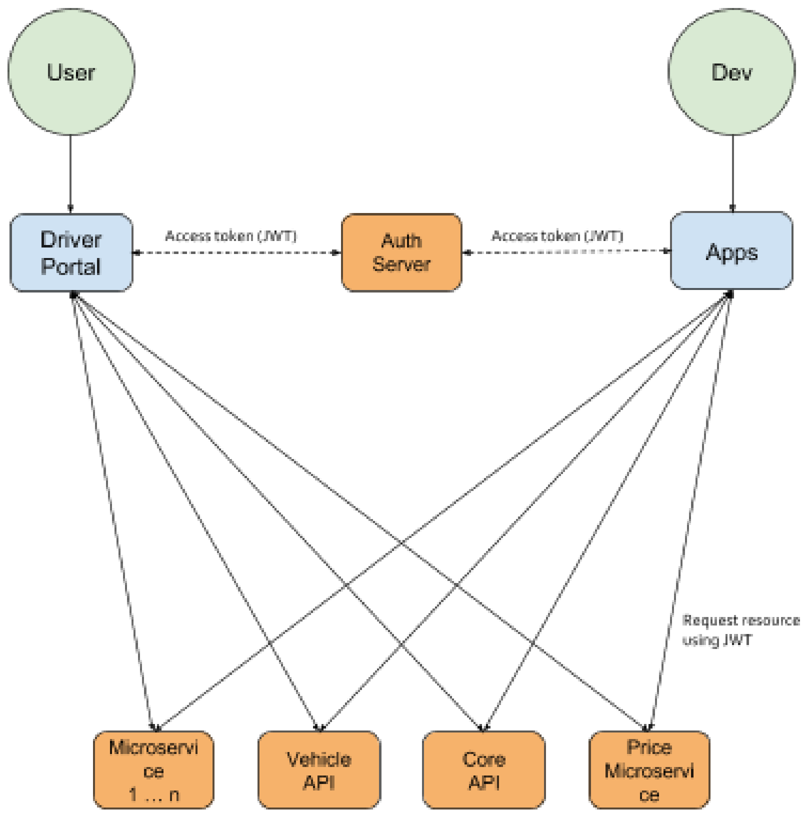

Fig. 3.4 OAuth with stateless JWT token requests

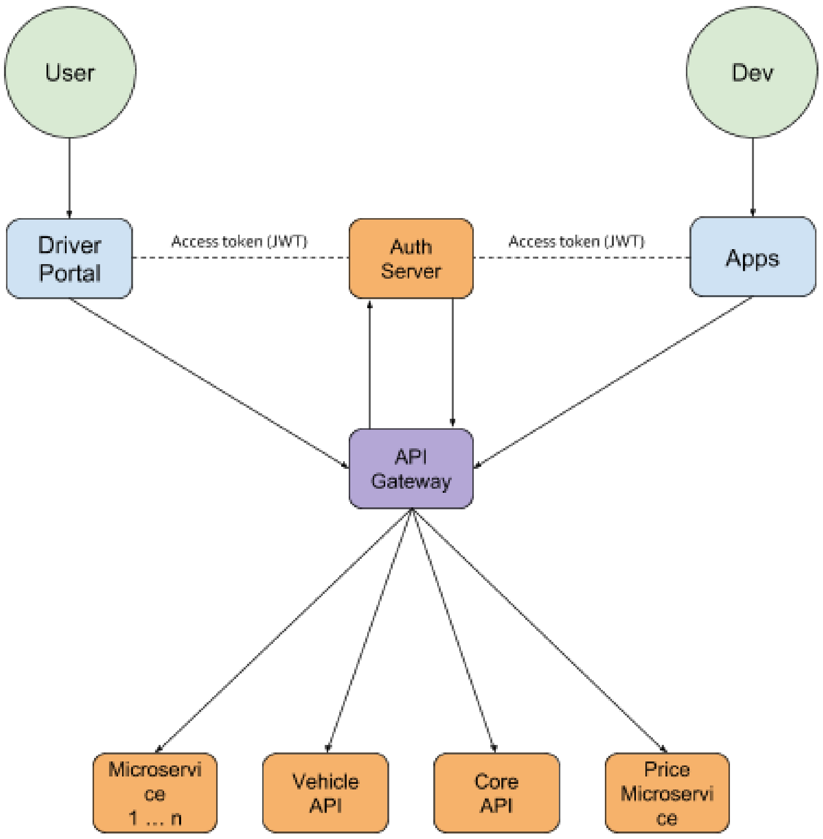

Fig. 3.5 API Gateway

3.4.5 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.

3.5 Suitability of Methods and Techniques

kks32: Write chapter about methods and technologies

- Node JS, PHP, MongoDB, MySQL, Microservice, Loopback, Graph QL,
- Mocha, Buddy-Works, Circle CI, Typescript, Chai, Functional Programming

As with most technological decisions, whether a microservice is desired depends on the problem that it meant to solve. When should a system be integrated or separated? It is a careful balancing between efficiency of satisfying dependencies keeping a separation of concern, and duplication of functionalities.

3.6 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.

References

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

[2] J. R. Herring, "Implementation standard for geographic information - simple feature access - part 1: Common architecture," May 2011. [Online]. Available: http://portal.opengeospatial.org/files/?artifact_id=25355

[3] (2004) Geographic information – simple feature access – part 1: Common architecture. [Online]. Available: https://www.iso.org/standard/40114.html

[4] J. R. Herring, "Implementation standard for geographic information - simple feature access - part 2: Sql option," August 2018. [Online]. Available: http://portal.opengeospatial.org/files/?artifact_id=25354

[5] (2018) Postgis 2.4.5dev manual. [Online]. Available: https://postgis.net/docs/manual-2.4/using_postgis_dbmanagement.html#PostGIS_GeographyVSGeometry

[6] (2018) Mysql 5.7 reference manual - geometry class. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/gis-class-geometry.html

[7] (2011) Iso/iec 25010:2011. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en

[8] 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

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

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

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

List of figures

2.1 LatLngSphere 7
2.2 Square 11

3.1 Architecture 16
3.2 Schema 18
3.3 Architecture 22
3.4 Architecture 23
3.5 Architecture 24

List of tables

1.1 Fixed Prices 2
1.2 Planning 6