Phase I - Pregame

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

subject Afstudeerstage bij TaxiID

year 2018

Author: Stefan Schenk
500600679

TaxiID
05-02-2018 te Medemblik

Tutor: Willem Brouwer

Hogeschool van Amsterdam
Software Engineering

Index

1. Introduction

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

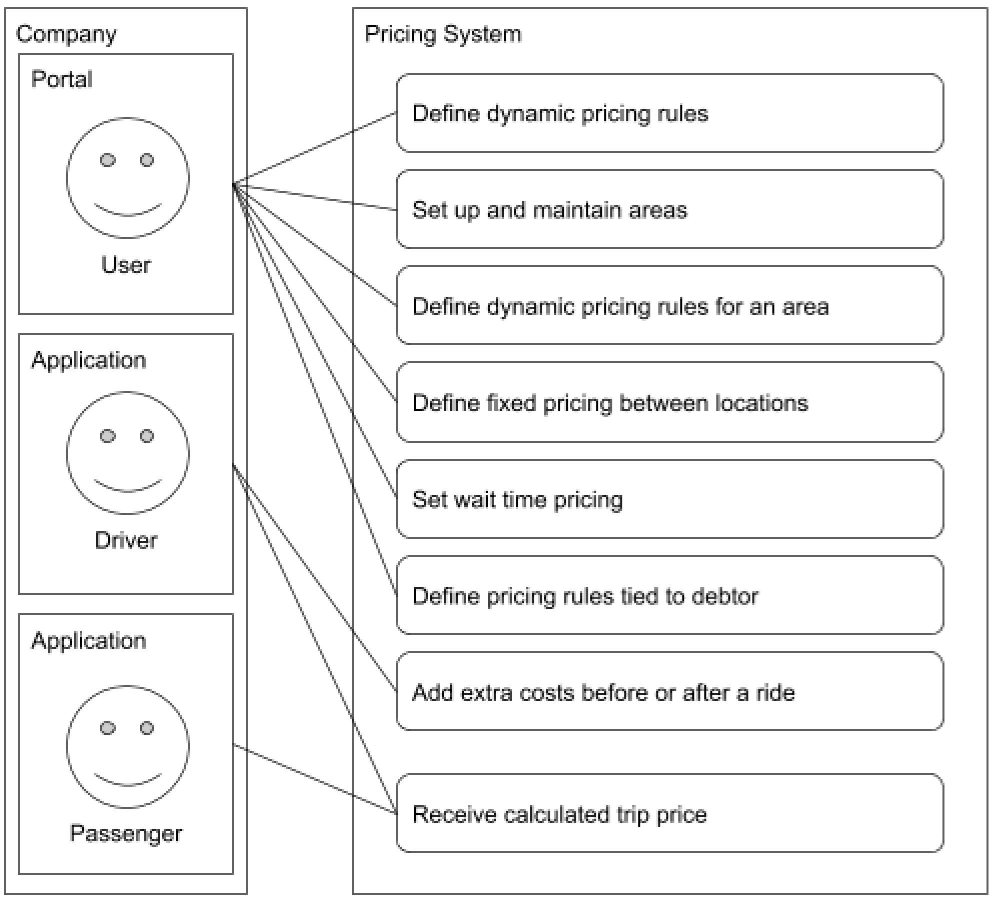


Image 2.4.1 - Use case diagram.

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.

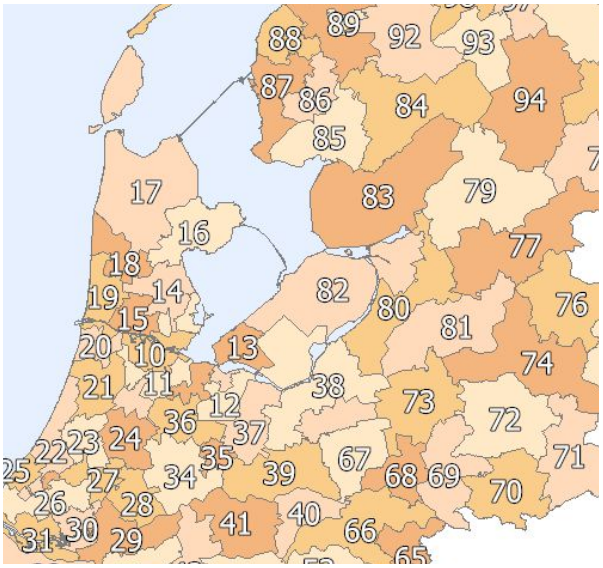
3.2. Functional Requirements

3.2.1. Defining an Area

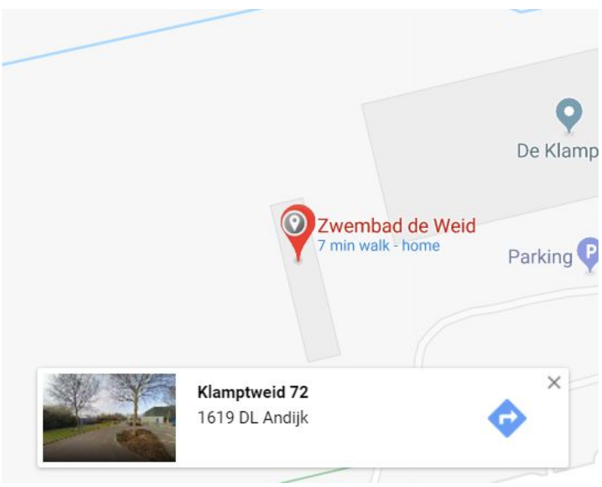
As most FR's depend on the term "area", it is a top priority to define what an area is. It's important to define locations in an unambiguous way so that no mistakes can be made like: selecting an area that is called the same. In some third-world countries, zip codes are not available, and area names 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.

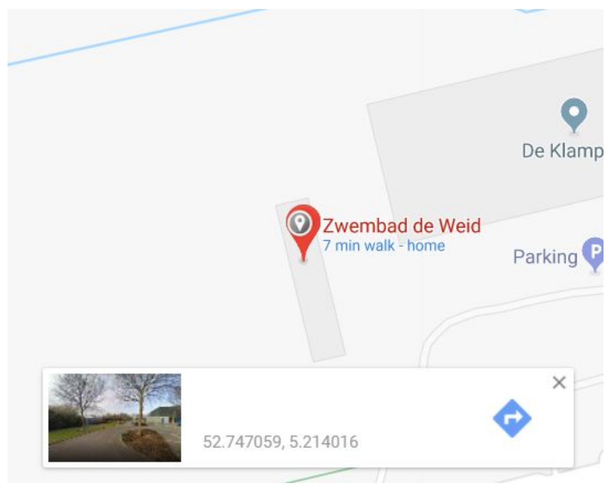
An area is a collection of 3 or more coordinate pairs on a geographical map. This definition of an area is precise, unambiguous and easy to use in compare in computer programs. A single point may match another single point if it's the exact same point. A point may be sitting on top of a line or is contained 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.

<table>
<tr><th>Countries with advanced zip code systems</th><th>Countries without zip codes</th></tr>
<tr><td>


Collection of zip codes to define an area
</td><td>


Polygon on a map to define an area
</td></tr>
<tr><td>


A single point defined by street,street nr, zip code city name
</td><td>


A set of gps coordinates with a range to define a single point
</td></tr>
</table>

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.

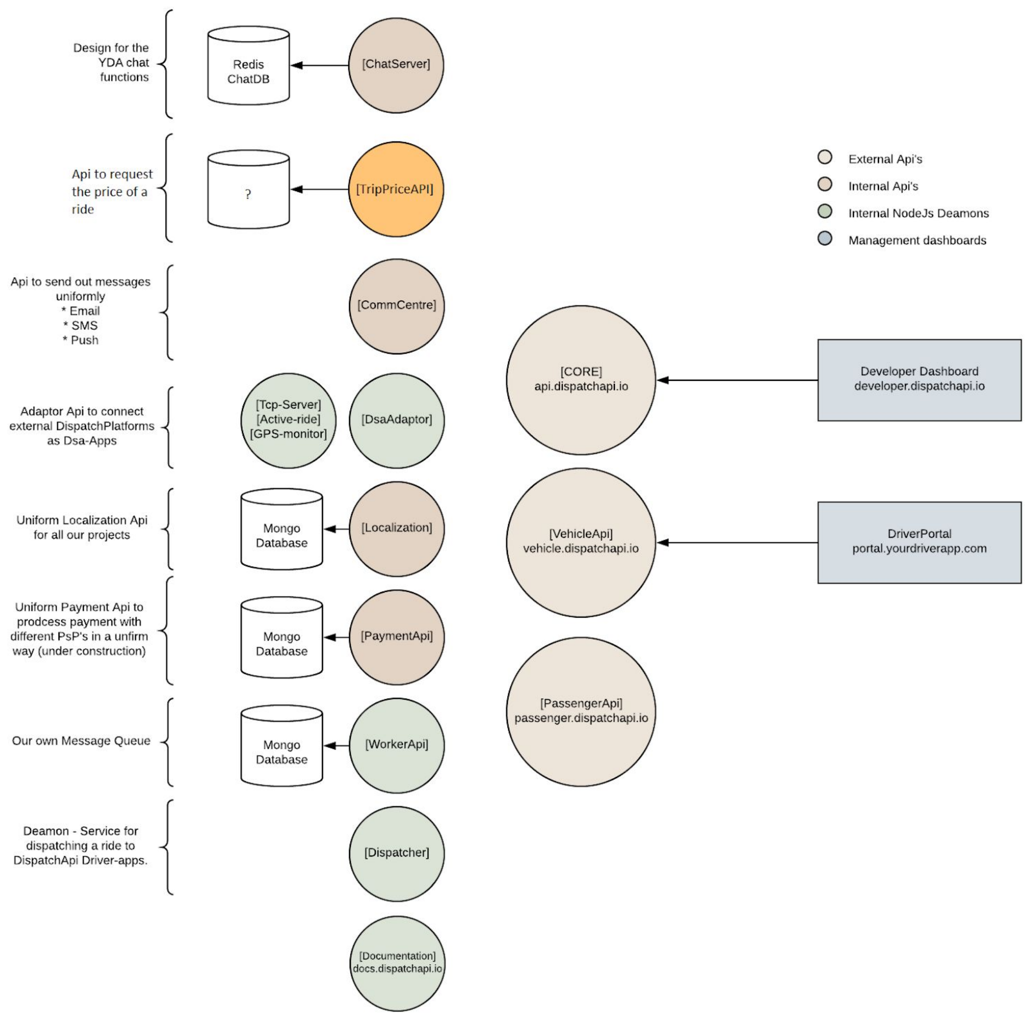


Image 3.3.1 - Current system architecture of the dispatch api.

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.

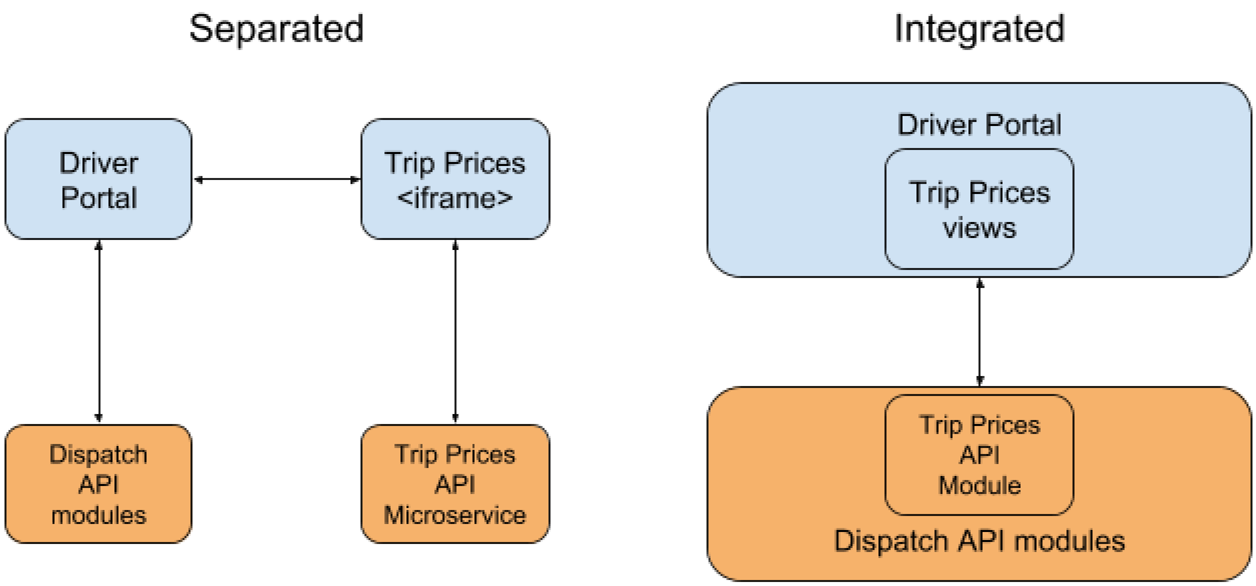


Image 3.3.2 - Separated and integrated frontend and or backend.

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:

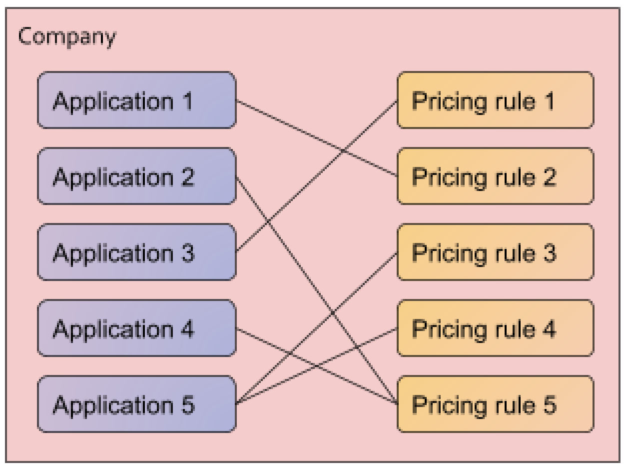


Image 4.1.1 - Company with applications and pricing rules.

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.

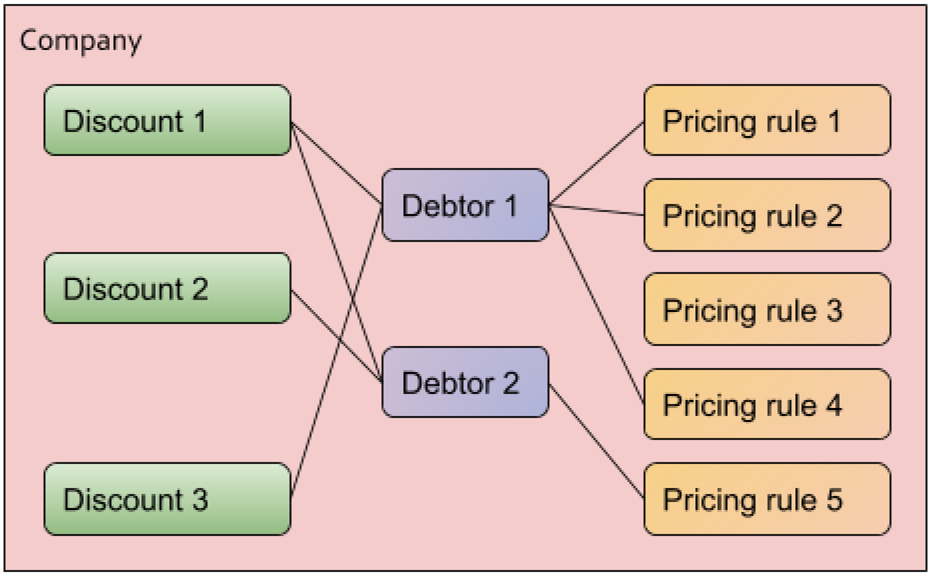


Image 4.2.1 - Debtors and the relation with pricing rules & discounts.

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.

<table>
<tr><th>Frontend</th><th>Backend</th></tr>
<tr><td colspan="2">Improves progressiveness of the entire architecture by incremental modernization steps.</td></tr>
<tr><td colspan="2">Improves maintainability by separation of concern.</td></tr>
<tr><td>Brings the advantage of including the application in any portal in the future.</td><td>Improves testability by having small subsystems that can be isolated and tested while other systems can be relied upon.</td></tr>
<tr><td>May introduce a technical difficulty of presenting the view correctly into the portal.</td><td>May require extra http calls between services.</td></tr>
<tr><td>May hurt the visual style.</td><td></td></tr>
<tr><td>Separation introduces a slight overhead because two separate views must be downloaded.</td><td></td></tr>
</table>

Table 4.3.1 - Pros and Cons of separation.

After discussing the proposal to segregate this project from the existing Dispatch API, it is advised to implement the backend as a microservice, not as a module within the existing system because the only downside that was listed is trivial if the services are running on the same server. From the viewpoint of this project, it is also advised to integrate the frontend in the existing portal

4.4. Authentication and Authorization

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

4.4.1. Proposal oauth 2.0 refactor

There exists a protocol to have a single source of authentication called oauth [3], which allows third-party apps to grant access to an HTTP service on behalf of the owner of the resource, or by allowing the third-party application to obtain access on its own behalf. This protocol solves the problem of having different implementations and tokens for authentication within the architecture.

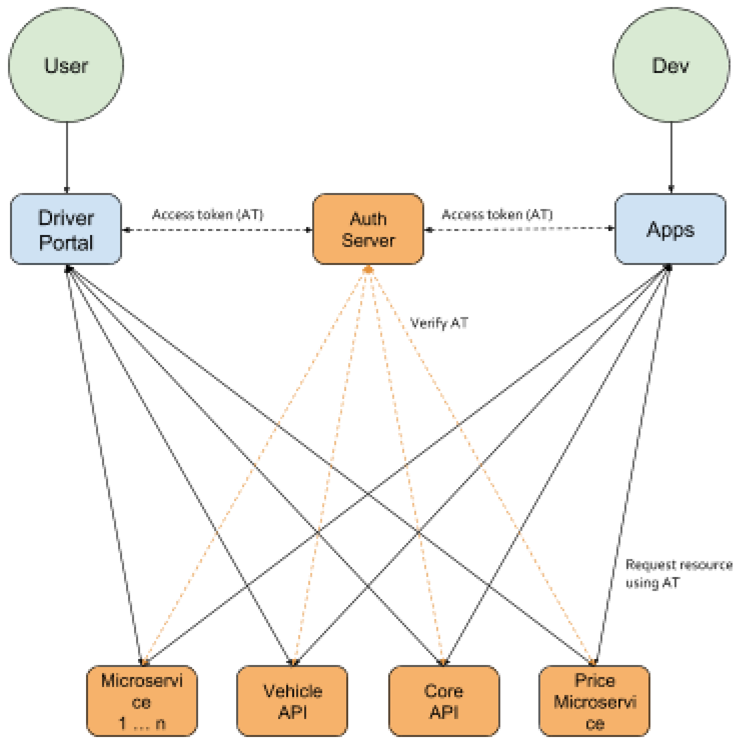


Image 4.4.1.1 - OAuth requests where tokens are verified by Auth Server.

4.4.2. Jwt token format proposal

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

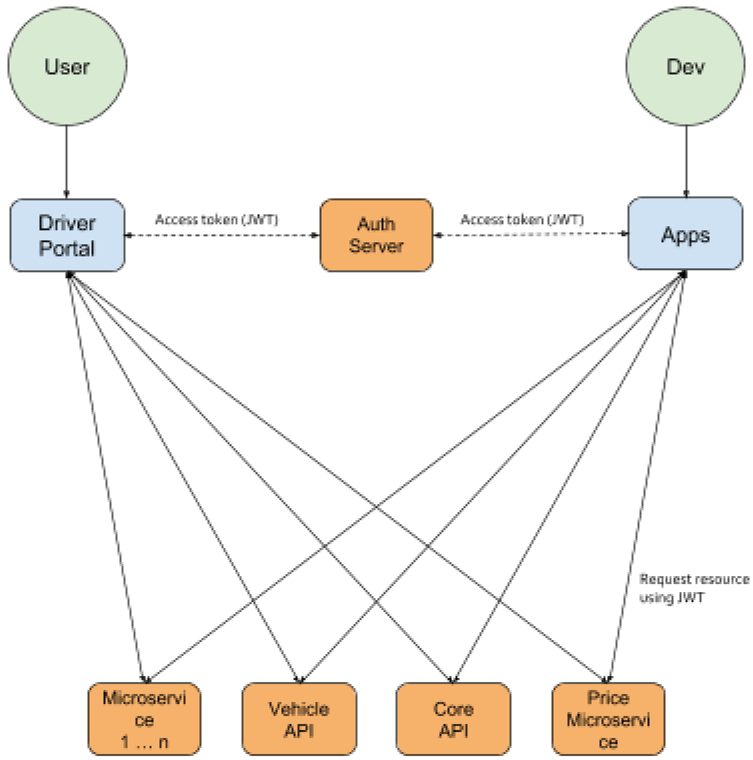


Image 4.4.2.1 - OAuth with stateless JWT token requests.

4.4.3. Proposal API Gateway

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

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

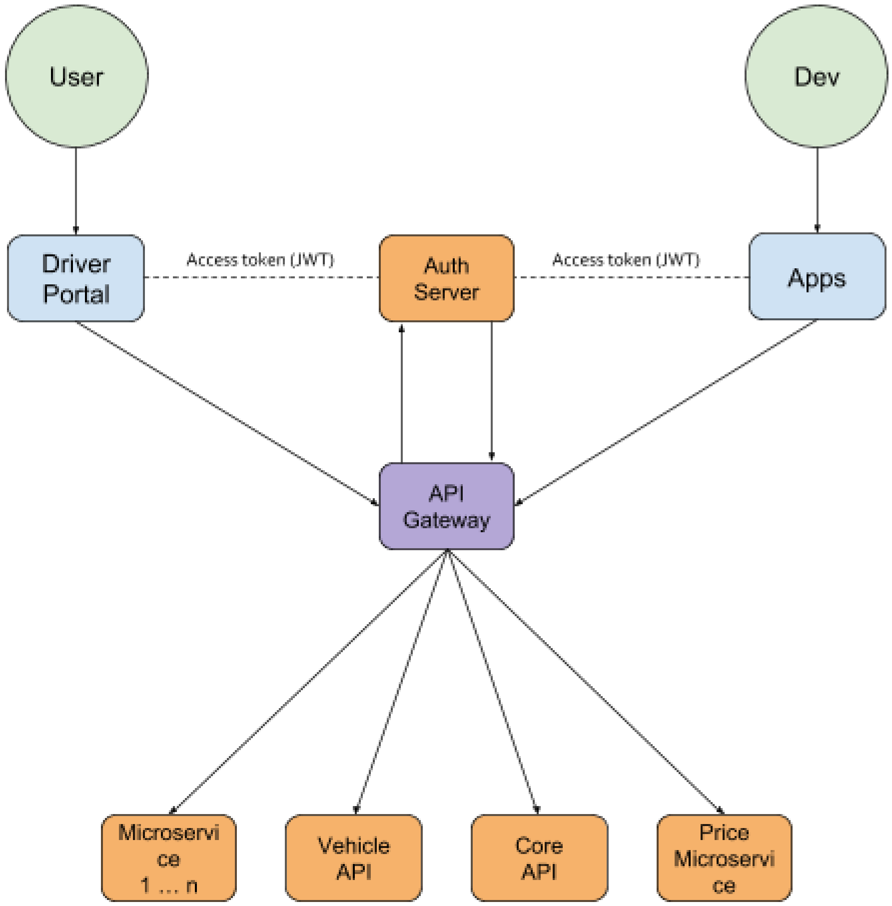


Image 4.4.3.1 - API Gateway.

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

4.5. Database

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

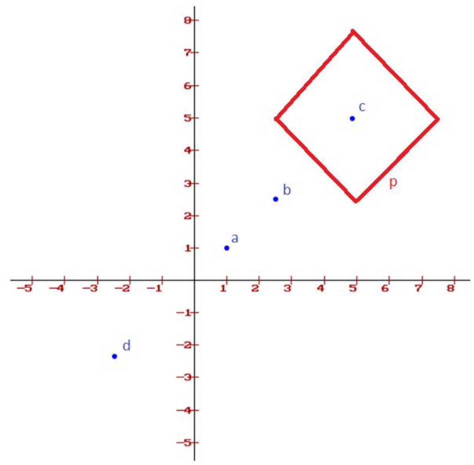


Image 4.5.1 - Four Points and one Polygon p containing Point c.

4.5.1. OpenGIS Compatible databases

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

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

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


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

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

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

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

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

4.5.2. OpenGIS Incompatible databases

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

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

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

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

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

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

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

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

4.5.3. Performance and Clustering Trade-offs

Agarwal & Rajan state that NoSQL take advantage of cheap memory and processing power, thereby handling the four V's of big data more effectively, but lack the robustness over SQL databases [10]. The report dives deeper into spatial queries and concludes that their tests suggest that MongoDB performs better by an average factor of 10, which increases exponentially as the data size increases, but lack many spatial functions that OpenGIS supports.
Although improvements have been made [11] after the cited paper Schmid et al. 2015 [12] was published.
The team argues that clustering is much easier in MongoDB, which may be important in the future when the company grows. As the required functionalities exist in both SQL and NoSQL, it is beneficial to opt for MongoDB for its performance and alignment with the teams experience. Although if robustness is desired, or extra GIS functionalities required, SQL should be taken into consideration.

4.6. API

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

4.6.1. Required Endpoints

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

4.6.2. Express VS Loopback

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

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

4.7. Database Schema

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

4.7.1. Relational Database

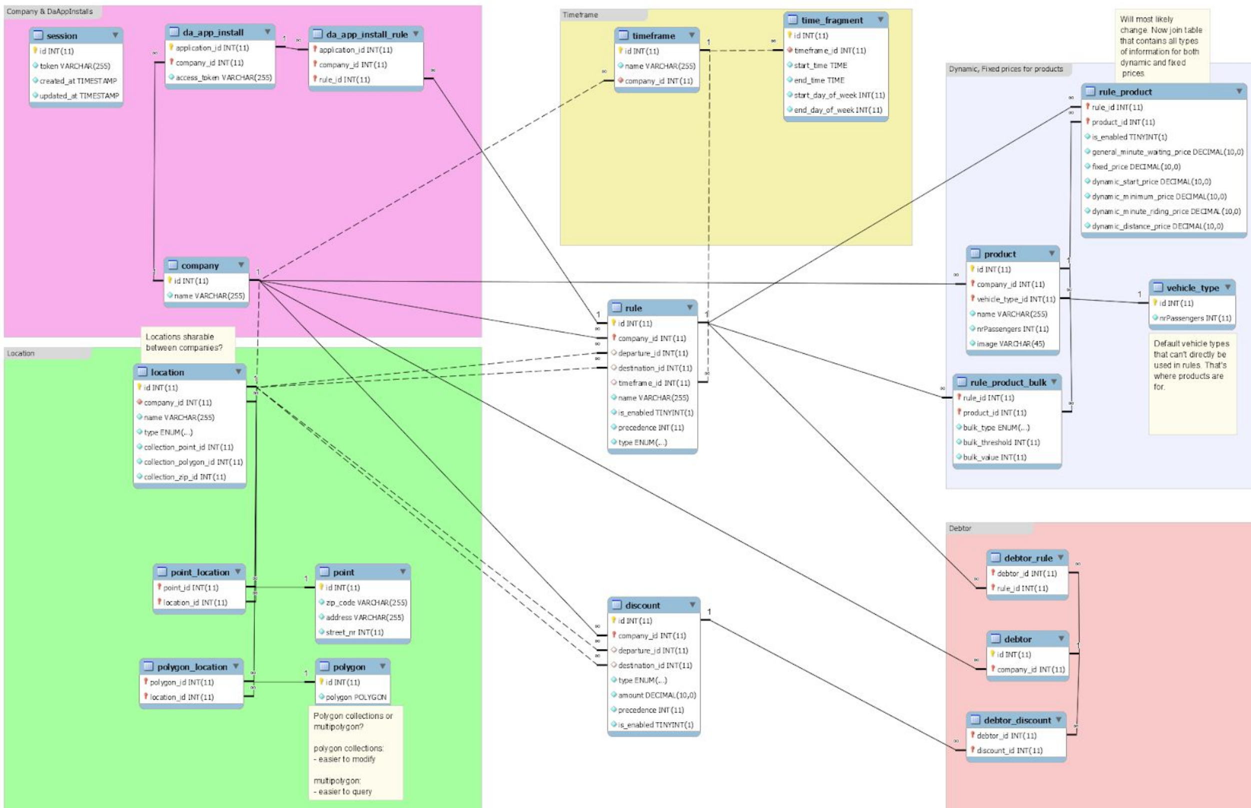


Image 4.7.1.1 - Rough schema for a relational database.

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

4.7.2. Non-Relational Database

```
// Application document
// with embedded settings
{
 _id: <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].