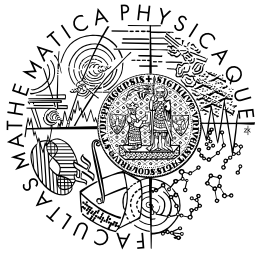

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Lukáš Březina

Taxi service back-end

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	doc. RNDr. Tomáš Bureš, Ph.D.
Study programme:	Softwarové a datové inženýrství
Study branch:	Databáze a web

Prague 2018

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Dedication.

Title: Taxi service back-end

Author: Lukáš Březina

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Tomáš Bureš, Ph.D., Department of Distributed and Dependable Systems

Abstract: Nowadays services like Uber start to surpass taxi companies in comfort of transport. The goal of this thesis is to create a back-end part of the application, which will increase the taxi services efficiency and give the users new and more comfortable ways to use them.

Keywords: Taxi Ruby on Rails

Contents

1. Introduction

The field of personal transportation has been revolutionized in a past few years. People in the metropolises got used to order the taxi via phone application. Nowadays it is almost standard to have all the information about the driver, vehicle, estimated time of arrival and price before making the the order. Also the service during the order process has evolved a lot. People are used to see the arriving driver's location in real time and also the estimated time of arrival to their drop off location during the whole order process.

However in the smaller cities is situation very different. None of the big companies operates in the smaller cities so there is usually no other way how to order the taxi than via phone. Phone lines are often overloaded during the peaks and customers are unsatisfied. Providing this new way of ordering would have massive impact on the company competitiveness.

In this thesis we created the back-end part of the application for order management and processing. We focused on the taxi companies from the smaller towns. We cooperated with the company from Mladá Boleslav, which provided us the insights from their business and described their work-flow. Keeping this know-how in mind we created the system.

1.1 Motivation

Besides the customer comfort increase in smaller cities this application reduces the operating costs of the company. Being dispatcher at taxi company is very stressful and low paid job. The main working hours are at night. These conditions lead to high staff fluctuation. Dispatcher is responsible for distributing the orders between the drivers and estimating the arrival times. When the dispatchers are changed often, they have no experience with the estimations which leads to fatal failures and highly unsatisfied customers. Having the algorithm for distributing orders between drivers and estimating the arrival times leads to stable more stable estimations which can be improved during the time. This reduces the skills the dispatcher needs to have for the job - now its job is just to communicate with the customer and insert the orders into the system.

Second advantage of having such system in the company is that they can process more orders with the same amount of dispatchers. Each order phone call takes two minutes on average. If they want to process more than thirty orders per hour they have to hire the second dispatcher.

1.2 Existing solutions

As we mentioned before, there are already existing solutions for this problem. From the most known companies operating in Czech Republic are Uber, Taxify and Liftago. They all provide the same feature - allowing the user to order the taxi via the application - but with a different idea in mind. The difference is that their customers can not order their services via a phone call.

This approach is not suitable for our case, because the company depends on

the users who order the taxi exclusively via phone call. Also our company has most of the profit from the scheduled large distance orders. These orders they want to manually split between their loyal drivers.

1.3 Goals

co práce bude obsahovat a co ne - na vyšší úrovni. Bude obsahovat api, které bude přístupné nějakým způsobem, z objednávek bude řešit obecně to a to... řešit objednávky od zákazníků, dispečerů, algoritmus pro přiřazování objednávke. Součástí práce není frontend - dělaný jiným člověkem.

poznámky

1.4 Outline

Dopsat

2. Requirements

In the first part of the chapter we focus on the requirements for the wholesome application solution and related business requirements. In the second part we specify the technical point of view.

2.1 Business requirements

Začít s nějakou vizí, jakým způsobem jsme se k těm informacím dostal, že jsme to rozdělili do nějakých kategorií a pak popis těchhle kategorií.

poznámky

Our goal is to define and formalize the process of ordering the taxi in the company. Based on the results we create the API which allows the clients go through this process.

Second goal is to design the scheduling system which distribute the orders between the drivers automatically. The main goal for the scheduler is to provide the customer detailed information during the order process and minimize the waiting time for the taxi. After the first estimation system should change the estimations as little as possible. Scheduler must support processing the orders with specified time of pick-up. The most important is to assure that the driver arrives precisely on time in this case.

2.2 Technical requirements

Our first requirement is to have the back-end and front-end strictly separated. This allows us in the future having more different clientswhich can be connected to our logic. With this architecture we can easily integrate ordering system into common applications like Facebook Messenger or Slack and do not relay on custom application only.

špatné pojmenování - client používán jako zákazník všude jinde

To nahoře není moc dobrý requirement - nco co se dá testovat, např. musí to být použitelné z mobilu i z webu.

poznámky

well documented vyhodit - nesmysl, psát jen požadavky, jakože bude více klientů, kteří s tím budou komunikovat. To samé bezpečnost - https není třeba zmiňovat.

internal requirements - frontend dát kdyžtak až nakonec. Nedávat i to s OS - dát třeba do requirements na development process

Napsat hlavně EXTERNÍ requirementy

Our API must be well-documented and easy to use so the front-end part can relay on the information in documentation. The communication with API must be encrypted - we should use the HTTPS protocol.

Because we have other developers working on the front-end side, we should keep the installation process of the back-end as simple as possible. We should be able to run it on various operating systems - specifically latest version of Windows, Ubuntu and MacOS X.

The stack we use for back-end application should be also prepared for scaling in the future and keep the downtime during the deployment at minimum.

3. Analysis

specification =¿ analysis Based on the business requirements we decided that the back-end part of the application must take these five divisions into account: Customers, Employees, Vehicles, Orders and Notifications. In the rest of the chapter we are describing features that these modules should have.

3.1 Customers

Customers are uniquely identified by telephone number. We also want to store about each of them these information:

- ID
- Name
- Note
- Fraud status

With Customer entity we are able to do these operations:

- Create and confirm
- Update
- Destroy
- Recover password
- Login and logout
- List favourite locations
- List all customers and show specific customer

3.1.1 Operation create and confirm

There are two ways how customer can be created. It is either directly through registration or indirectly by creating new order.

Directly registered customers are created in exchange for telephone number, password and optional name. With this type of account customer can later login with provided password. Application must verify given telephone.

Indirectly registered user is created during creation of new order for telephone number, which doesn't belong to any existing customer. This customer type is just envelop for the purpose of tracking information and statistics - mostly for better customer support. This account type can not be used for authentication. Indirectly registered user can be directly registered later without any difference to normal direct registration.

3.1.2 Operation update

Customer can update only it's own name and password. Employees are able to change any customer's name, note and fraud status.

3.1.3 Operation destroy

Destroy the customer can be invoked by that customer or the administrator. Orders made by that customer are not deleted - the order has no customer then.

3.1.4 Operation password recovery

In case of lost password is customer able to recover it. At first customer asks for the recovery with its telephone. In return it receives recovery token via SMS. This token is valid for 5 minutes. With this token and telephone number can new password be set. Customer is able to ask for token resend - which will invalidate last token, generate new token and sends it via SMS.

3.1.5 Operation login and logout

With login operation we receive login token in exchange for telephone number and password. We send this token with each request to be authenticated. Customer can login if and only if it is directly registered and confirmed. Logout just invalidates the session - customer must log in again to be authenticated.

3.1.6 Operation list favourite locations

Our application must provide list of customer favourite places. In the front-end part of the application should be this implemented in the part where customer chooses its pick-up and drop-off location.

Main priority is to provide list of these places fast = less than one second. Most important are first five returned places and the most likely places based on current conditions should be among them These places should be ordered with respect to the given location (in reality current customer location or pick-up location when choosing drop-off). It should also take into account whether customer chooses pick-up or drop-off. These recommendations should be based on customer's order history and respect start - finish relation of the orders.

Let's imagine customer that has two routes - it often goes from pub to its home and sometimes goes from its home to gym. Application then should return its home as first item when looking for drop-off recommendation from unknown pick-up place. Application should also return gym as first item when user is asking for drop-off recommendation with pick-up at home, even though pub is much more frequent place in its history.

Show the favourite locations list of a specific user is permitted only for that specific user or any employee.

3.1.7 Operation list all customers and show specific customer

Our API must provide information of the customers created in our application. Show specific customer's data is available only for the customer itself or any employee.

List all the customers is available for administrator only, because the list of the customers is one of the most valuable asset of the taxi company and we don't want to provide it for the staff. Of course the employee could get all the customers by going one by one via operation show, but it takes more time so this is for our purpose enough.

3.2 Employees

There three types of employees - administrators, dispatchers and drivers. Employees unlike customers are identified via email. We store these information about each one:

- Email
- Name
- Photograph

In our application we also have information about the employees shifts - whether it is at work or not. For drivers we also process current locations and their order queues.

Operations on employees are almost the same as operations on customers. Operations differs mainly in permissions.

- Create and confirm
- Update
- Destroy
- Recover password
- Login and logout
- List all employees and show specific employee

3.2.1 Shifts

Employees could be in three statuses:

- available
- unavailable
- pause

Available means that the employee is on site and can handle orders. Unavailable is when it is not at work. Pause status is there for the situations when the employee knows that it won't be available for a while but wants to finish its orders.

Switching directly from available to unavailable should be only in cases of emergency, e.g. driver has flat tire and cannot continue. Employees will be instructed not to do so for better customer experience.

Employee is available to list the history of its shifts and administrator is available to list shifts for all the employees. Changing shifts (available statuses) are employees allowed only for themselves.

3.2.2 Driver locations

Each driver from the shift start until the shift end sends at regular intervals its location. Driver's current location is able to set only driver itself. Get the last driver's location is can anyone, so the front-end can display the current location of arriving driver even for anonymous customer.

3.2.3 Driver order queues

Each driver has a queue of orders that are assigned to it. Administrator can see all the queues, driver can see only its own queue.

3.2.4 Operation create and confirm

Employee can be created by administrator only. In exchange for the email, optional name and image the confirmation email is sent to the employee. Then the employee is by clicking the link in email redirected to front-end page, where it fill its password. The link contains confirmation token which will the front-end together along with the optional name and image send to our application.

3.2.5 Operation update

Employee can update its password, name and image. Administrator can besides these fields change also employee role.

3.2.6 Operation destroy

Only administrator can remove the employee from the application. When the employee is removed, all the shifts and driver queue is removed too. Orders associated with the employee remains in system but the corresponding employee field is removed.

3.2.7 Operation recover password

Recovering the forgotten password is exactly the same as in the customers case - except the reset password token is sent via email in link to the front-end.

3.2.8 Operation login and logout

Only difference in these operations between the employees and customers is that the employees login with email instead of telephone number. Everything else is the same.

3.2.9 Operation list all employees and show specific employee

Show all the employees or specific employee can only administrators and dispatchers. The rest (drivers, customers, anonymous users) can see only drivers who are on shift.

There is also different attributes which are shown for different people. Public attributes that anyone can see are id, name and image of the employee. Email, roles, and other attributes like timestamps of creation or update are available only to employee itself, dispatchers or administrators.

3.3 Vehicles

Taxi company has the vehicle fleet we want to have in our system too. Each driver's shift starts with selecting the vehicle driver will ride in, so the customer can see and choose the car that fit its needs.

For each vehicle we have these information:

- name
- internal vehicle number for taxi company used for communication
- plate
- image
- how many customers can fit in - required
- whether the vehicle is available for driving or not e.g. is temporarily in the car repair shop

These operations with vehicles our application supports:

- Create and update
- Show all and specific
- Destroy

3.3.1 Operation create, update and destroy

Only administrators can create, update and destroy company's vehicles. They can manipulate all the specified information. When the car is deleted, all the corresponding shifts or orders will have set the null value instead of the vehicle.

3.3.2 Operation show all and specific vehicle

Administrators can see all the vehicles, others can see only the active ones.

Operations show to anyone all the attributes besides the internal vehicle number and the availability. Employees can see all of the attributes.

3.4 Orders

Order is key entity in this application. Each order must go through process from creation to successful finish or cancellation.

There are two types of orders. The first one we call scheduled - customer wants the taxi to arrive to its location at specific date and time. In this type of order arriving at the specified time is crucial. The other type is when customer just need taxi and sooner it arrives then better.

Order can be come from two sources - dispatchers and customers directly.

About each order we would like to have these information:

- id
- status
- driver who takes care of it
- vehicle by which it is processed with
- pick-up and drop-off location coordinates and addresses
- passenger count
- note
- contact telephone in case the customer is ordering for someone else
- estimated price
- whether the assigned driver can not be changed (is explicitly chosen)
- VIP (just internal flag for the taxi company)
- flight number - in case that the order is to/from the airport
- customer
- assigned dispatcher
- date and time when the customer wants the taxi to arrive to the pick-up location (scheduled pick-up)
- source - whether it was created by dispatcher or directly by customer via front-end application

Also with order we want to know track these time parameters. In parenthesis are the terms how the times will be referenced in the whole thesis and application:

- created time
- application estimate when the driver starts arriving to customer (start est.)
- when the driver started arriving to customer (start)
- estimation when will the driver arrive to the customer (arrived time est.)
- actual time when driver arrived to the customer - (arrived time)
- estimation and actual time when the driver has picked up customer and starts driving to the drop-off destination (picked-up time est., picked-up time)
- finish time estimation and actual finished time(finish time, finish time est.)

Our goal is to have as much data about the order as possible, so we can later make statistics and analysis based on them.

These actions are needed for the orders module:

- show all / specific order
- list driver arrivals
- create order
- defraud and process
- confirm by driver
- refuse by driver
- arriving
- change arrive time
- arrived
- customer not on its place
- picked up
- change drop off time or location
- finished
- fraud
- my orders for dispatcher
- cancel

Because the order process is very complex, we decided to sum it up in the one illustration. This scheme displays all the order statuses (green), actions that can be done with the order and this application implements them (violet), time parameters we track (blue) and notifications which are sent in these actions (blue).

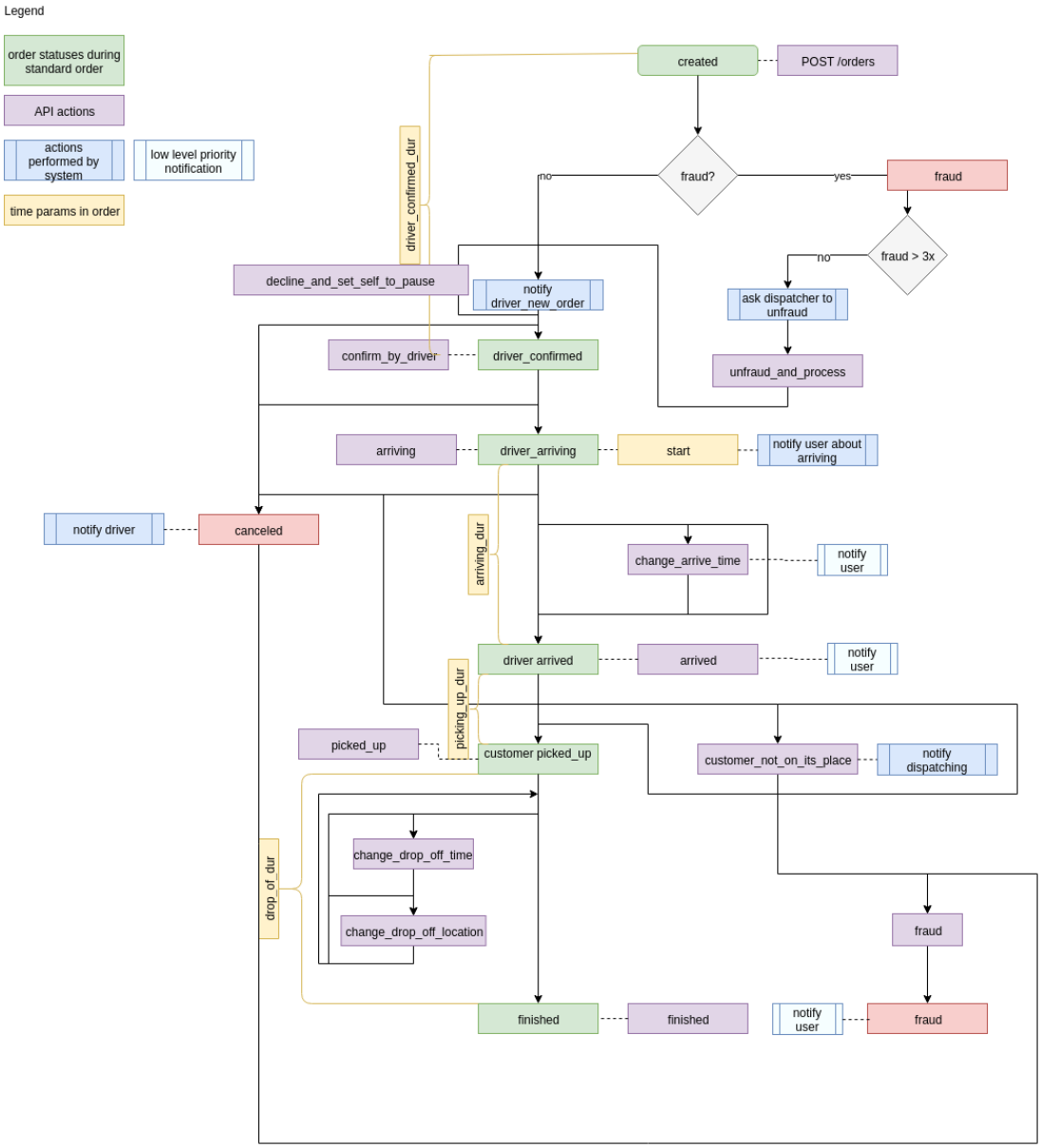

Figure 3.1: Order process scheme

In next subsections we describe in detail all the actions, their prerequisites, conditions and outputs.

3.4.1 show all / specific order

Employees can see all the orders, customers can see only orders which they have made (even via dispatcher).

We show specific attributes to the specific user types. Anyone (e.g. anonymous customers) can see order status, created time, arrived time est., finish time est., driver and vehicle assigned to it.

Customer whose order it is can get also coordinates and address of start and finish, passenger count, note, contact telephone, estimated price, whether assigned driver is selected explicitly, VIP flag, flight number, arrived time, scheduled pick up and source.

Employees can besides all of the attributes above see assigned dispatcher and all the tracked times, estimations and original estimations.

For showing all orders we also show total order count for the specified query.

Orders support basic filtering. We can get orders which are created since and until as parameters. We can also get only scheduled orders. Orders can be filtered also based on their status - we can set multiple statuses we want to show.

Orders are paginated - we can set via parameters which page to show and how much orders per page.

3.4.2 List driver arrivals

This action must return for the start and finish location list of the available drivers with their cars including the estimated arriving times to that location. This information then dispatcher tells the customer in case the order is via dispatching. In the other case customer can see it directly in its app.

It also takes driver parameter for the case that the customer wants the specific driver and only checks for the arrival time.

This list is provided for anyone - even anonymous users.

In case there is no driver available for such conditions it should return no available drivers result.

3.4.3 Create order

Order can be created two ways. Either it is via dispatcher - customer calls the taxi company and the dispatcher will make the order.

If the customer is marked as fraud, new order is automatically marked as fraud one, thus it does not continue in process and the creator is informed about that. If the number of fraud orders in history for the customer is less than three, one of the dispatchers on shift get the notification whether this customer should be forgiven and order processed. If the user has more than three fraud orders, dispatchers won't even get the notification about defrauding.

Both the customer and dispatcher can explicitly choose a driver who takes care of the order.

Order will be assigned to the driver who can be at the pick-up location earliest. Scheduled orders are meant to be assigned to drivers manually by dispatchers.

If no driver is available for the specified parameters, it must return error.

After this action, order status is *created*.

3.4.4 Defraud and process

This action is called by dispatcher as response to defraud notification - when it wants to forgive the customer frauds in history and wants to process that order.

If the customer is forgiven, it is marked as non-fraud and order is normally processed. Customer is from that moment on considered as standard non-fraud user, so it can create subsequent orders without limitation. The forgiveness is though not definite - possible next fraud order will renew original customer's fraud order count and add one fraud more.

3.4.5 Confirm by driver

When order is assigned to driver, driver confirms via this action that it knows about it and is able to take it. Until then, the arrive time estimation is very rough. This action can be performed only by the assigned driver and only if the order status is *created*.

This action changes the order status to *driver_confirmed*

3.4.6 Refuse by driver

Driver also can refuse order. We suppose that this will happen only in emergency cases. If driver refuses order, it is removed from its queue and passed to the next available driver. If there's none, order is marked as cancelled.

To refuse the order, its status must be *created* and it can be done by driver assigned to the order only.

3.4.7 Arriving

Driver can have more confirmed orders in its queue. When it starts to go to the customer, it calls the Arriving endpoint. This change the order status to *driver_arriving* and recalculate the time of arrival to customer. In this moment it should very precise estimation - driver should be slowed down only by the traffic which is included in the Google Maps API. Thanks to that we send the SMS to the customer with the estimated arrive time in this moment.

The order status must be for calling this action must be *driver_confirmed* and it can be called by driver assigned to order only.

3.4.8 Change arrive time

Driver can correct the estimated arrive time based on its experience or unexpected complications during the journey.

Correct the arrive time can only driver for his own orders which are in the *arriving* state.

3.4.9 Arrived

This request is sent when driver arrives to the pick-up place. This changes order status to *arrived* and updates following times estimations, because now we have precise moment of arrival.

It can be set by the assigned driver only and only when the order status is *arriving*.

3.4.10 Customer not on its place

In case that the driver can not get in touch with the customer this action is called. It sends the notification to the dispatching and the dispatching take care of the situation and communicate the problem between the driver and the customer. This can situation can be resolved only by the to ways. Either is customer found and picked up or it is not and the order is marked as fraud.

This action can be called only when the order status is *arrived* and by the driver assigned to the order.

3.4.11 Picked up

In a point when the driver picks up the customer and is ready to ride to drop-off location is this request sent. Because now we know the picking up time precisely, drop off estimations can be recalculated. Also this action changes the order status to *customer_picked_up*.

This action can be called only when order status is *arrived* and only by driver whose is the order.

3.4.12 Change drop off time or location

Customers often don't know what they want, so in our application driver assigned to the order can change in this point the drop off time and also the location. This change leads to the recalculation of the estimated times.

3.4.13 Finished

Order is marked by driver as finished when he successfully handled whole order and is ready to go to another customer.

3.4.14 Fraud

Order can be marked as fraud from the *driver_arrived* and *cancelled* order statuses. Mark order as fraud can only driver assigned to the order. Marking as fraud resets the customers fraud count back to all the fraud orders it has.

3.4.15 Cancel

Order can be cancelled by the driver assigned to order, dispatcher or the customer whose order it is. In case that the customer is anonymous it can be cancelled by anyone - because we can not distinguish separate anonymous users.

Only valid order statuses from which can order be cancelled are *'created'*, *'driver_confirmed'*, *'driver_arriving'* and *'driver_arrived'*.

3.4.16 My orders for dispatcher

Each order is assigned to some dispatcher. These dispatcher than take care of the customers and in case of the problems must be able to see that problem and handle it with the customer. For such purpose there is this endpoint. Dispatcher

can see all its order with important details such as the actual time estimations, original estimations, customer number, assigned driver and so on.

This endpoint shows only orders which are not finished, cancelled or fraud and the results are paginated.

3.5 Notifications

Our system must have a way how to send messages to the users on specific actions. As described in orders section, there are three types of notifications:

- driver has new order
- customer is not at pick-up location
- new order from fraud customer

Each type contains specific data. Driver has new order contains the order id, customer not at pick-up location contains driver id and name, customer id and telephone. New order from fraud customer contains order and customer id.

Notification system is passive = users call our *list notifications* action in regular intervals. When user resolves notification, it can mark the notification as resolved.

3.5.1 Operation list notifications

This operation returns all the notifications for the current user. The list is paginated via *page* and *per_page* parameters. It also supports filtering only unresolved notifications.

3.5.2 Operation mark as resolved

Notification can be marked as resolved only by the person to who it belongs.

4. Technical Analysis

In this chapter we want to take a look on specific frameworks and tools that we have used for the whole solution and what led us to decide that way. In the first part of the analysis we describe things related to our back-end application software stack - for example what frameworks and tools we use and why we have made such decision. In the second part we focus on the whole server stack and how we manage to run all the services on it.

4.1 Back-end application software stack

For our application we decided to use Ruby on Rails[1] framework written in Ruby[2]. Our asynchronous jobs are handled via Sidekiq[3]. We decided to use PosgreSQL[4] as our main database engine. We also run Redis[5] as it is required database for Sidekiq.

4.1.1 Ruby on Rails

There are many frameworks in which could be this type of application written equally well - for example ASP.NET(C#), Spring(Java), Laravel(PHP), Django(Python), ExpressJS(JavaScript).

Here are some advantages and disadvantages of Ruby on Rails which has led us to choose it.

Advantages:

- Simplicity and expressibility of Ruby - optional parenthesis, return keyword, no semicolons, combination with functional programming
- Strong Convention over Configuration influence - you have strictly given where to place models, controllers, how to name classes, database tables etc. and you are forced to do it that way. It may seem limiting at first but it brings to project clarity and most of the times it gives you good way to solve your problem without reinventing wheel
- plenty of tools built in - from the routing and security through development-testing-production configurations to the highly sophisticated ORM
- global repository of libraries (Ruby Gems) - most of them in very good quality with clear documentation and test covered
- We are using it for 3 years, so we know proven libraries and ecosystem

Disadvantages:

- it is more difficult to set it up than PHP

[1] Ruby on Rails framework main page `https://rubyonrails.org/`
[2] Ruby language main page `https://www.ruby-lang.org/`
[3] Sidekiq wiki page `https://github.com/mperham/sidekiq/wiki`
[4] PostgreSQL database main page `https://www.postgresql.org/`
[5] Redis database main page `https://redis.io/`

- impossible to use standard web hosting
- small base of programmers knowing Ruby
- efficiency compared to some framework

Add citation

4.1.2 PosrtgreSQL

We decided to use PostgreSQL, because it is open source and unlike MySQL it supports natively storing JSON, arrays and it has many plugins - for example for storing geo data. None of these features we use in our application now but why not to have this possibility when we would like to optimize something or extend it. Since we use Rails ORM (ActiveRecord), choice of the database is not so critical - we can migrate later to other database.

4.1.3 Sidekiq

We need job processor to handle sending emails, SMS, and calculating favorite places for customers. There are many job processors for Ruby[6]. We decided to use Sidekiq, because it is long-standing project focused on efficiency and we have used it before. However for project with requirements like these, any of the processors would handle it equally well.

4.2 Tools stack

For documenting our API for all the front-end applications we use Apiary. The tool which features we decided to use for installation and deployment all of our services is Docker with Docker-compose. As our reverse-proxy and HTTPS certificates manager we decided to use jwilder's ngnix-proxy image set. As error catcher and alerter for all of our running apps we decided to use Errbit. For the server monitoring and future alerting we decided to use uschtwill's docker_monitoring_logging_alerting image set.

Include links to all libraries

4.2.1 Apiary

We needed tool, where could be all the possible requests to API with proper responses well-documented for front-end developers. We decided to use Apiary mainly because it was tool designed and developed in Czech republic, thus I knew it before and knew that it fullfills all of our requirements. Even it allowed us to make API mocks running on their server for free, so frontend developers could design and set up their interfaces while we could still work on our implementation details.

Our whole server stack looks like this: graph of all the services

[6] Job processors comparasion `http://api.rubyonrails.org/classes/ActiveJob/QueueAdapters.html`

4.2.2 Docker and Docker-compose

Our goal was that each frontend developer would have its own local version of backend on which them could develop. To achieve it, we had created tutorial how to install Ruby, Rails, PostgreSQL and all the SW stack described before. This was not a good approach at all. Despite the problems with installation (different versions of Ruby, Rails) it took almost 3 hours to set up one machine. Also we were not able to make stack work on Windows which was a big issue for our Android developer. With such experience we decided to use Docker and Docker-Compose to handle the stack.

Docker is a platform for developers and sysadmins to develop, deploy, and run applications with containers. A container is launched by running an image. An image is an executable package that includes everything needed to run an application–the code, a runtime, libraries, environment variables, and configuration files. A container is a runtime instance of an image–what the image becomes in memory when executed (that is, an image with state, or a user process).Docker Inc. [2018b]

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.Docker Inc. [2018a]

Using this tools allows us to set up container for each service (application, sidekiq, database, etc.) and run smoothly on any operation system using only one command - *docker-compose up* with insignifficant performance change[7].

4.3 Docker proxy & ngnix

Because we want to run multiple websites on our server (back-end & front-end applications, monitoring & error sites), we have to deal with reverse proxy. During research we have found out, that for that purpose was made set of Docker images - jwilder's ngnix-proxy, which does exactly what we needed - including automated management of HTTPS certificates.

That is another reason why use docker. Propper set up of reverse proxy with HTTPS on production is now matter of few hours without previous experience with ngnix nor Let's Encrypt and adding other site is just the matter of starting new container with three environment variables.

4.4 Errbit

Our goal was to have service where we could see unexpected failiures of all our apps. This service should notify us whenever this failiure occures. We should be also able to get at least basic info when, where and on what environment that failiure occured.

All these requirements fulfills Errbit, which is open-source alternative for more known Airbrake. Errbit is also written in Rails, so it is close to our stack. Also it has standalone ready-to-use docker image.

[7]from our observations during development

4.5 Monitoring

Because our server stack is composed of more than ten services, we want to have the logs from all of our services merged to one place, where we could analyse them. Also we want to see current system status and health from web browser. During the research we have found stack of docker images for logging and monitoring by uschtwill.

The logging stack that we use consists of

- ElasticSearch - database engine for storing and searching logs
- Logstash - aggregates logs using docker gelf driver and pushes them to ElasticSearch in propper format
- Kibana - frontend for exploring ElasticSearch database, predefined dashboards, searches etc...

Choice of this stack for such task was confirmed by Peter Havelka on Hradec Kralove barcamp, who said, that they are using exactly same stack for 16M rows of logs per day with no problems and that it helped them a lot with analysis and general overview over their services. Havelka [2017]

5. Solution design

analýza =¿ návrh řešení

In this chapter we would like to explain the thinking process before and during the implementation of the back-end application. We are going through individual parts from the back-end application requirements, explaining problems associated with them and revealing what possibilities we had to solve them and how we decided in the end.

5.1 General problems

5.1.1 Authentication

In our application we had to deal with authentication for customers and employees. There are many ways how users can be authenticated in API. We ended up with token authentication - in each request clients must set the Authorization header with the token. Client can get the token in exchange for correct credentials.

Another part of the authentication we consider in the analysis is at least the basic security during the manipulation with auth data and flawless verifying token sending.

Token authentication details

We decided to implement it on our own, using only Rails helpers for generating and verifying secure tokens. As you can see in 5.1, token is generated during the login action and then returned in body of the response. All the following requests must have this token in it's header to be authenticated.

Our application store just one token per user. That implies the user can be logged in from one device at time only. Having more valid token for users would let into several problems with invalidating them in case of log out and also this approach has one advantage. If someone reveals user credentials and log in with them - user will know that in the moment it tries to use the app, because all it's requests would be unauthenticated.

In Ruby on Rails there exists very good gem for authentication - *devise_token_auth* [1] which we half-implemented at first. However we decided to not use it in the end. Main reason for not using it was - at the time of writing the authentication part - very unstable and badly documented front-end library part. More details about problems we came across during the analysis/implementation in the next paragraph.

First reason for not using the gem was, that it uses email as main default authentication field. For customers we wanted to use the telephone number as the main identifier - including the SMS confirmation and password recovery (explained later), which would led us to rewrite the most of the gem's controllers and models anyway and as bonus we would have to integrate it with the existing parts of gem.

[1] `https://github.com/lynndylanhurley/devise_token_auth`

Figure 5.1: Auth token scheme

Second reason was the complexity of the whole authentication process. This gem would bring more secure but also much more complex way of authenticating to our project. Once the client gets the token from the login endpoint, a new token is generated and returned with each following request. Using this approach we also have to solve the batch request problem. Imagine that the client sends three requests to the server at once. Of course all of the three requests must have the same authentication token - because the client doesn't have the new token until the response arrives. Thus the server must accept all of those three requests as authenticated. Besides that they also must agree which response has the correct new auth token - responses don't have to come in order in which they were sent. This complicated process was implemented in the back-end gem and couldn't be easily changed. The front-end part of the library seemed to be implemented, but after the three weeks of trying and founding several bugs in the library - even without successful login - we gave up using it.

In conclusion if we had wanted to use the gem on backend we would have to understand the whole gem auth process in detail and specify it for the front-end. Front-end would then have to write the whole complicated solution on their own.

We came to conclusion that the complexity, that would bring this library to our application is not worth the better security and stability we would gain from this. Using HTTPS on both server and client makes the token theft little bit harder - so lack of this generation process with each request is not a big deal. Also the only really sensitive accounts for identity theft are the employees one's and we are in personal contact with them, so we would know about misuse or potential hack and we could provide the safer solution later.

Basic security

As we mentioned in last paragraphs, our token solution is not perfect from the security perspective. Besides the token (which could be easily rewritten more securely in the future) we tried not to take the security of our application lightly.

First of all, we don't store passwords in plaintext. We use Rails integrated feature *has_secure_password*[2], which uses BCrypt hash function.

Thanks to the Ruby on Rails framework we also excerpt passwords from the logs and we are not vulnerable to SQL Injection[3].

Verification tokens sending process

We send the verification tokens via emails and SMS during the whole authentication process. These actions are not instant (API request, SMTP request) and in Rails we can easily create asynchronous workers. We decided to have these functions handled as Sidekiq workers. In exchange for some time spent on configuration we don't block webserver threads during account creation/recovery and also we have ability to automatically resend the message in case of the third party or network failure.

5.1.2 Secrets

In our application we have to keep several API tokens, database passwords and other sensitive information. We store those secrets in files which are not tracked in version control. Those files are then loaded by the Docker, which set them as environment variables. In the whole application we read these sensitive information from them.

5.1.3 Authorization

Almost every endpoint has it's own rules of who and in what circumstances can do such action. There are two main groups of users - employees and customers. Employees are then divided into three groups - administrators, drivers and dispatchers. There are just two types of customers - registered and unregistered. Each visitor is one of these types.

In each request we have to solve specific condition whether current user is able to do this action or not. Having the all these conditions directly in controllers would make the controllers difficult to read. Also these conditions could change in the future and have changed during the development several times. This led us to have the authorization conditions separated in the different part of the application.

We want to avoid reinventing the wheel so we have chosen to use for such purpose *Pundit* [4] gem.

Another option we looked into was *Cancancan* [5] gem. They both have long maintenance history, are actively developed in the time of writing and satisfies

[2] `https://api.rubyonrails.org/classes/ActiveModel/SecurePassword/ClassMethods.html`

[3] `https://en.wikipedia.org/wiki/SQL_injection`

[4] `https://github.com/varvet/pundit`

[5] `https://github.com/CanCanCommunity/cancancan`

all of our conditions for the authorization. Cancancan is better suited for applications with complicated views, because it provides more helpers for checking authorization in them. Also its architecture is more general thus it is better optimized for more complicated permission management. That results in slightly more complex permission definitions. On the other side pundit is more lightweight solution with very simple architecture and permissions definition files. Because our permissions are not complicated very much and we wanted to have them written as simply as possible, this was the main reason why we chose Pundit over Cancancan.

5.1.4 Pagination

With growing data some of the requests (e.g. order index) could return thousands of items. This would be huge waste of resources, so we have to limit the count of returned items per request and enable pagination. For that purpose there exists gem *Kaminari*[6] which adds to our models methods for quick filtering and retrieving data. We just define pagination request parameters and pass them to this library. Also thanks to the global configuration we can limit the maximum number of results per page.

5.1.5 Rendering views

Even though whole view layer is just about displaying simple JSON objects, we use whole separated view layer of the Ruby on Rails framework with the help of the *Jbuilder* gem[7].

In the beginning of the implementation we thought that we don't have to have the view part separated from the controller. For the simple entities such as vehicles it was sufficient. When we started to have more complicated requirements for the views - such as displaying different attributes for different user roles, most of the controller's code was the view logic which mystified the controllers.

Jbuilder allows us to have view code separated from the controllers and provides us simple syntax to extract desired fields and parts from our variables to JSON.

5.1.6 Images

Our employees and vehicles have assigned images. We had to figure out, how to get the image from the client application via the REST to our application.

During the analysis we ended up with two options how to implement it. First is the multipart html requests. . Second option is to encode the image into Base64 on front-end and then send it in a request body as JSON field. This field is then decoded on the server side and saved.

citace https://tools.ietf.

In the end we decided to go with the second option. On both front-end and back-end we have the libraries for processing images this way so the implementation was easier on both sides. Downside of this solution is that we can not

[6] `https://github.com/kaminari/kaminari`
[7] `https://github.com/rails/jbuilder`

transfer much data this way. However we have only one image for each employee and vehicle, this solution is sufficient.

For the back-end side we decided to use *Carrierwave::Base64* [8] gem. Besides the out-of-the-box Base64 encoding of specified attribute which we need, it provide us simple API for the storage settings such as file names. Also when we would like to later switch our storage to use some cloud storage provider as for example S3 by Amazon[9] it would be just the matter of changing few lines in configuration.

5.2 Customers

Use email as main distinguishing field is kind of standard in web authentication. We came to the conclusion that we should use as our identifier telephone number. Despite the standard and the the consequence of this decision - lack of any easy to use library for authentication in Rails. Reasons which led us to this decision:

- During the order process we must be able to contact customer immediately in case of emergency, so we need the customer's phone anyway.
- Customers are going to register mostly from their phones. That phone can receive SMS for sure - not everyone has direct access to his mail from phone.

5.2.1 Create and confirm

Besides the authentication and params problems described in general problems section we also have to to deal with the telephone verification.

give links

We decided to use verification via SMS code.

Registered telephone number must be verified. Before the customers can do so, they must go through telephone number confirmation process as follows: Customers receive SMS with registration token. This token is valid for 5 minutes and customer can ask for resend. Resend will invalidate last token, generate new and send it. Confirm is made with provided token and telephone number.

Based on requirements we decided to split these functions into three API endpoints - *create*, *confirm* and *resend_confirmation*.

5.2.2 Password recovery

Whole password recovery procedure is similar to the create account one. Based on specification we split the password recovery feature into two API endpoints - *password_recovery* and *reset_password_by_token*.

Calling first endpoint - password recovery - sends in exchange for the telephone number SMS to that telephone number with password recovery token. Then the customer can send request with this token, his telephone number and a new password - which if all the conditions are satisfied - will be set.

The token consists of 4 numbers and is valid for 5 minutes. This parameters we just picked as similar to the others services on the internet and of course we must be able to change them later if we discover that it's not good.

[8] `https://github.com/y9v/carrierwave-base64`

[9] `https://aws.amazon.com/s3/`

During the analysis we have noticed, that the first endpoint must except the bad request format always return success status. Especially, we can not let the client know whether the account with such telephone number was not found and neither that the SMS was not sent successfully. If we would return error code in such situation, someone could potentially get the telephone numbers for all of our customers.

Also we are aware that the SMS may not be the most secure way of verifying users[10] but we think that at our scale, potential losses for stolen customer's account wouldn't be crucial - there is no credit system or something valuable on the account. The worst case is that the attacker orders a taxi on victims name and thus that order will be fraud - which happens few times a week now anyway.

5.2.3 Favourite places

We decided to have an endpoint with four parameters, which will return the list of N recommended places ordered from the most to the least appropriate.

First is parameter is the maximum number of places we would like to receive, second is customer id for whom we want to have recommendations for, third is the location and the last - 'start' - parameter says one of the following:

- true = we want recommendations for pick-up places and the provided location are customer's current coordinates
- false = we want recommendations for drop-off locations and the provided location are pick-up coordinates

Our first problem to solve is how to handle the locations from the request. We suppose that the location which goes to our API is directly from the customer's telephone sensors, thus there's nothing like Example's restaurant official coordinates, which would client sent to us whenever he wants taxi from that restaurant. On the other hand, we would like to group all these locations near the Example restaurant into one place, so we can recommend it just once and there was enough space for other interesting places. We thought about using some kind of modified clustering algorithm but we ended up with conclusion that this would be in our case overkill. We came to conclusion that for our use case is enough to have defined distance constant and all the places around this place within the constant distance are considered as one place. We set this constant to 50 meters, because it seems like good compromise between inaccurate low-end phones GPS precision and the distance between two different places. Of course that we must be able to change this constant in the future if we discover, that it is too small or big.

Second problem was how to filter and return the list of places fast. For each user we have index of visited places with information on which we base our recommending algorithm. In this index we have following information about each place:

- coordinates
- list of items containing for each place occurrence in orders:

[10] `https://www.cnet.com/how-to/why-you-are-at-risk-if-you-use-sms-for-two-step-verificatio`

  - decimal number from 0 to 1 which says, how long ago was order with this place placed. 1 has user's last order and 0 has user's furthermost order .
  - time-stamp when was the order created
  - start = whether was the place in the order used as a start or finish
- list of corresponding places (drop-off for pick-up and vice versa).

For the given parameters in request, we go in index place by place and count the weight of it from the occurrences weights multiplied by constant if start/finish fits the desired direction. Index is also prepared for recommending places based on the order time (e.g. in the evening we go to pub, in the morning to work). We decided to limit the recommendation index for last 1000 orders for each user.

In our research we haven't found any ready-made solution for this problem, so we decided to come up with our own solution. Of course it could be more effective and recommendation could be done even smarter. During the development we came to this algorithm which satisfies all our specified metrics and thus we were satisfied with the application of it in the real world.

5.3 Order scheduler

The goal of order scheduler is to split new orders between the drivers in application. Besides that the system accepts other actions for manipulation with the order. These actions are then reflected to the corresponding driver queue and to all the orders affected by such actions.

First of all we have to deal with the distance calculation. All we know about the new order are just the coordinates of the pick-up and the drop-off location. We have to calculate the distance and the duration of the order. Each order consists of the two parts we have to calculate. First part is from pick-up location to drop-off. This part is constant for each order. Second part is the route between the driver and the pick-up location. This part differs for each of the drivers. For the distance and duration computation we decided to use the *Google Maps Distance Matrix API* [11] which allows us to get the distances between all the drivers and the order pick-up location in a single request. Also thanks to using this API we could use the existing Ruby Gem *googlemaps-services*[12] which provides us the Ruby interface for communication with that API. This API can also calculate the distance and duration with respect to the current traffic. Downside of this API is that each route calculation is expensive.

Second problem we are facing are the scheduled orders. Arriving at the pick-up location on time is the priority number one. Because of the business requirements scheduler does not assign these orders to a driver but the dispatchers do that manually. This implies that in our system could appear order collisions which must the system handle correctly. Also we have to take in to account high cost of each router calculation.

Based on the specification we decided that order scheduler system accepts these actions:

[11] `https://developers.google.com/maps/documentation/distance-matrix/intro`
[12] `https://github.com/amrfaissal/googlemaps-services`

- Add
- Change arrive time
- Cancel
- Change driver

In following subsections we analyse possible situations that can occur in each of these actions and how should the planning system handle them and respond to them. We display the situations using the diagrams described in the legend 5.2.

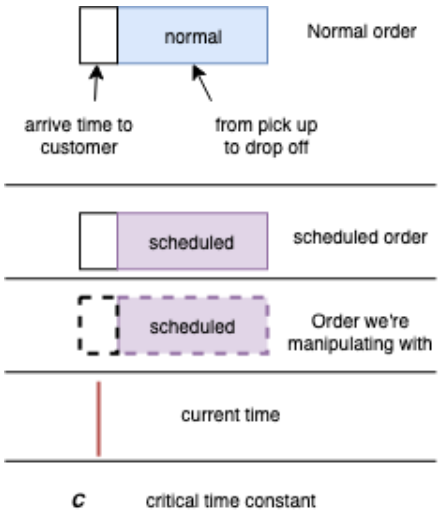

Figure 5.2: Order planning scheme legend

5.3.1 Add order

We distinguish two main categories of added orders - scheduled order and normal (not scheduled) order. At first we have to define who can be assigned the order, and then we have to define how to do it.

Order can be assigned to either all of the available drivers or to the one specific driver. By available driver we mean the driver who is on shift and is not in pause status. Then from this set of drivers we choose the driver whom the order will be assigned to.

As mentioned before, scheduler must ensure that if the scheduled order is assigned to the driver, it is able to get there right on the pick-up time. Here appears a problem we must solve - how to count the arrive time for a given driver to that pick-up location so we can let know the driver when to start processing the order?

First naive solution lead us to count the arrive time in the moment of adding to queue. In this point we know the last order finish location, so we could count arrive time from that point. The problem is that the order we are adding can (and probably will) be week or two after the time of arrive to that location. In order to persist the calculated arrive time we have to options. First one is to forbid adding

new orders before the scheduled order - that is obviously unacceptable. Second one is to during the insertion of some other order in between the last order in queue and our scheduled order count two arrive times - the newly inserted one and the scheduled one 5.3.

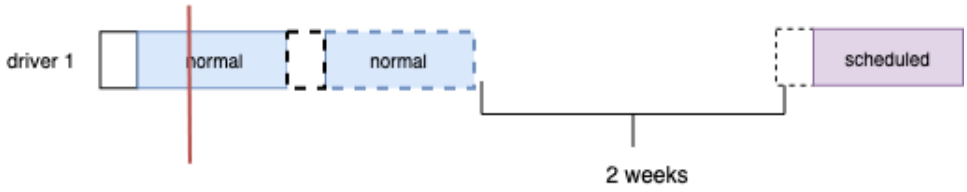

Figure 5.3: Scheduled order - naive arrive time calculating approach

Each route calculation is expensive and this naive approach doubles the costs of each processed order by the driver who has some scheduled order in the queue. Because these orders with two weeks ahead pick-up time are the most common use case in our taxi company we want to avoid this extra cost.

We observed that there must be some point, when the arrive time of scheduled order must be calculated. If we calculate to early, we raise the costs per order. In the other case if we calculate to late the driver could not be able to get to the pick-up location on time.

Our solution combines these two cases. Based on that observation we define the critical time constant C. Value of this constant is equal to the maximum arrive time to any location operated by the taxi company.

In our scheduling system then holds that each scheduled order in queue which has pick-up time less than C from the order before it or the current time, has calculated arrive time which we update with each change. If the distance between two orders is larger than C we do not count the arrive time.

This led to split the problem of adding order into seven parts:

- Normal order - counting arrive time
- Normal order - selecting driver
- Scheduled order - counting arrive time
- Scheduled order - combined with other scheduled
- Scheduled order - combined with normal order
- Normal order combined with scheduled
- Scheduled order - scheduling on critical time callback

Normal order - counting arrive time

First we focus on how to count the arrive time for the normal order. As we can see in 5.4 there are three states in which can queue be.

As in situation *1a)* if the last order in queue is a normal order, calculate the arrive time as time between the finish location of that order and start of the new order.

In case that the queue is empty as in *1b)* calculate the arrive time as time between the last known location + time for response to the order for the driver.

Third possible situation *1c)* occurs when there is ongoing scheduled order. Then count arrive time as the time between finish location of that order and start of the new order.

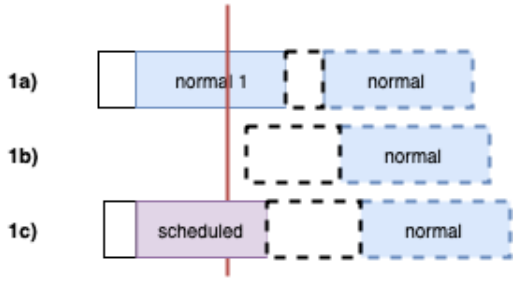

Figure 5.4: Normal order - counting arrive

Normal order - selecting driver

As described in specification, normal order must be always assign to the driver who can pick up the customer first. This rule demonstrates the example 5.5. The order will be assigned to the *driver 1* even though the *driver 2* is available first.

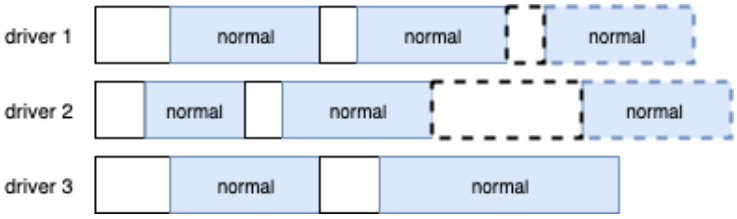

Figure 5.5: Normal order - counting arrive

Scheduled order - counting arrive time

If we add scheduled order to the driver's queue in which there is no other order within the critical time constant, we do not calculate the arrive time of that order5.6. We are allowed to do so, because the critical time constant C ensures, that if the queue stayed in the same state, the arrive time between the last location to the order start will be less than C.

Scheduled order - combined with other scheduled

In case that the scheduled order has other scheduled order within C from the calculated pick-up and finish time, we have to calculate corresponding arrive times 5.7. After the calculation we may encounter the orders collision. In such case the order can not be added to the queue and the order scheduler must throw an error.

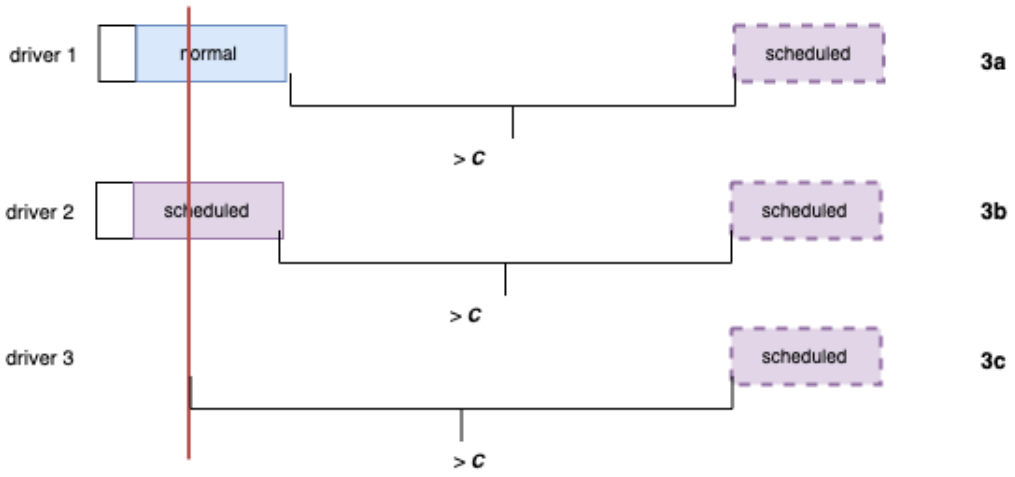

Figure 5.6: Scheduled order - counting arrive time

Scheduled order - combined with normal order

When the expected finish time is within the C from the inserted scheduled pick-up time, we must calculate the arrive time. In case that the the scheduled order is in collision with some normal order, scheduler must throw an error 5.8.

Normal order - combined with scheduled

Adding new normal order to queue could break the fact that each of the scheduled orders in our system has counted arrive time if the pick-up time is at least C from the finish time of order before it.

If this situation occurs, we must recalculate the arriving time of the scheduled order after it - accordingly with respect to the new finish location. This change can lead to collision. In case we encounter the collision, we start the process of adding the order again. In this second try we consider as the previously picked queue's finish location the finish location of collided scheduled order. Also the arrive time of the colliding scheduled order must be set back to the original value. 5.9.

Scheduled order - scheduling on critical time callback

As we know scheduled orders have not calculated arriving time when their pick-up time is within C from any other order finish in queue or the current time. In previous cases we described all the possible situations how can order be added before the scheduled order within the time constant. Last thing we have to cover is the situation when the current time exceeds a point from which is the the pick-up time of our order less than C.

This situation we would cover with callback which is called exactly in that point. If the order does not have arrive time calculated in that point, we calculate it from the last known drivers location and the order start location. 5.10.

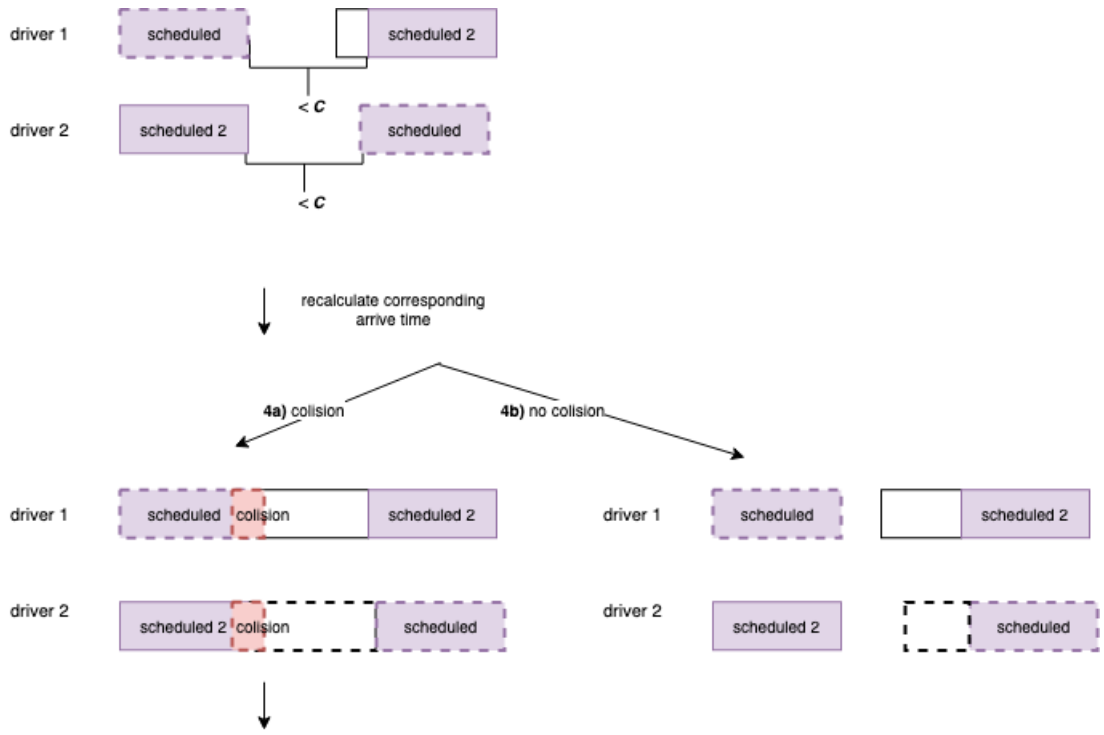

Figure 5.7: Scheduled order - combined with other scheduled

5.3.2 Change arrive time

Estimated order times can be manually changed by driver during the order. Average expected time change in such cases is within a few minutes. We can divide the change of time into two groups. In the first one the order will take longer after the change, in the second one the order will be shorter.

Longer order

When the order takes longer than it was expected to we just take all the normal orders after it until the first scheduled one and we move their start times accordingly5.11.

During this change there can occur collision with some scheduled order in queue. If the colliding order is normal order, we reschedule it the same way as we were adding it 5.12.

If the collision is between scheduled or ongoing order, we must shift the start of the following scheduled order 5.13.

Shorter order

When the time change makes the order shorter, we just simply change the starting time estimation accordingly for all of the normal orders after it until the first scheduled one 5.14.

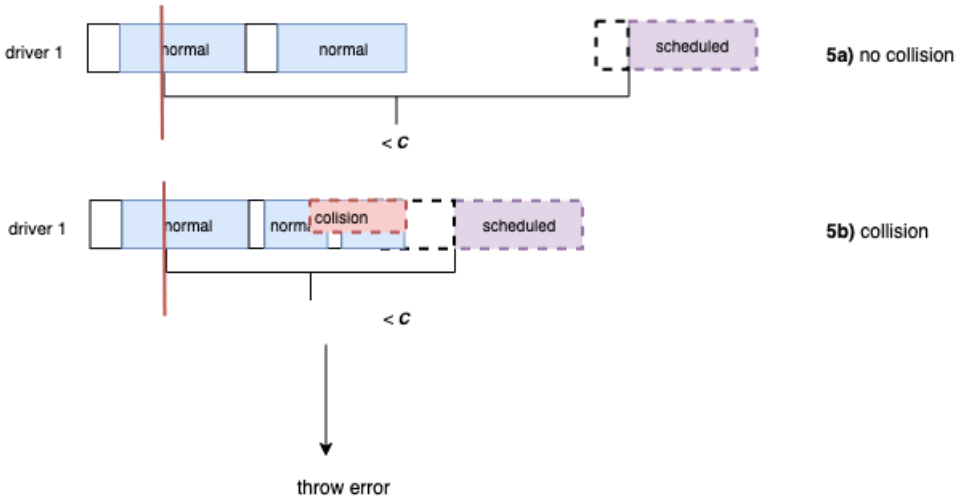

Figure 5.8: Scheduled order - combined with normal order

5.3.3 Cancel

When the order is cancelled, we must remove it from the driver's queue and reschedule each order after the cancelled order - except the scheduled one. By rescheduling we mean sequence of two actions - cancel and add 5.15.

We suppose that the order cancel is rarely called, so we do not pressure on the route calculation optimization as much.

5.3.4 Change driver

Changing the driver should happen even less than cancelling the order. In such case it is sufficient just to call the *cancel* action and *add* action with the specified selected driver.

5.4 Orders

For our application is critical to keep track of the order modification history. In case of unsuccessfully handled order we must be able to tell whether it was the scheduler failure or the mistake was caused by an employee or a customer. Also having the order time estimations history may help us in the future during the improvements of the scheduling system. We have found the *paper_trail* gem[13] which fulfils our needs and is easy to use and setup.

For the each order we have to keep track of the several time fields. In one half of the application we have to know and modify durations (arrive, waiting) and in the other half we have to know exact times (start, arrive to customer, finish). We have two options how to save such times. We can preserve timestamps for each event and the durations compute on demand. We decided for the second option - save the order start timestamp and duration in seconds for the rest. Thanks to this approach we can easily manipulate with the orders in the scheduler. Downside of this approach is that we can not easily filter and sort the orders via those calculated times.

[13] `https://github.com/paper-trail-gem/paper_trail`

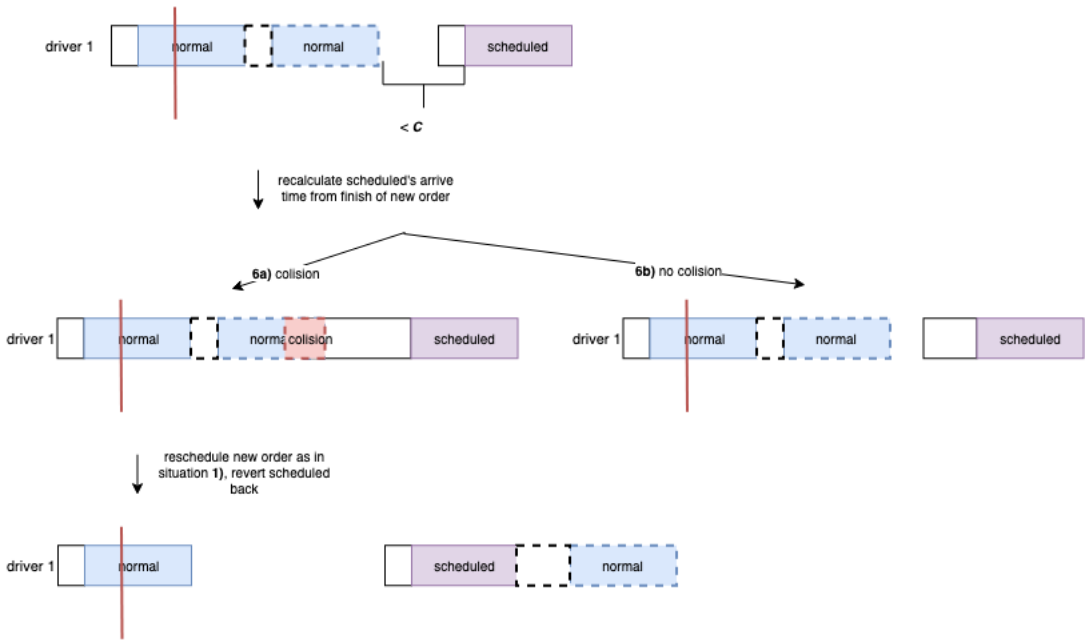

Figure 5.9: Normal order - combined with scheduled order

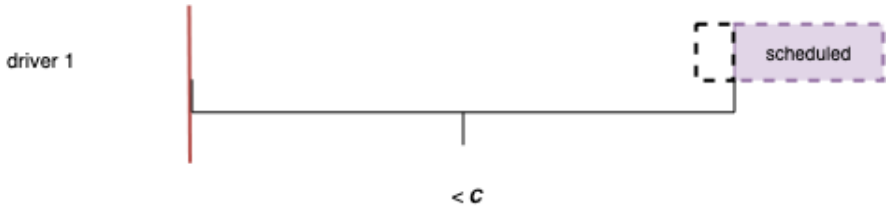

Figure 5.10: Scheduled order - critical time callback

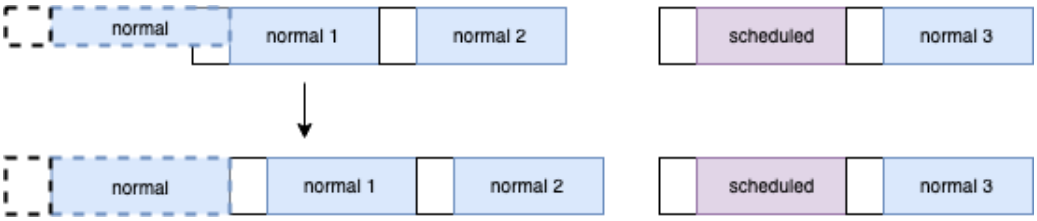

Figure 5.11: Change arrive time - longer order

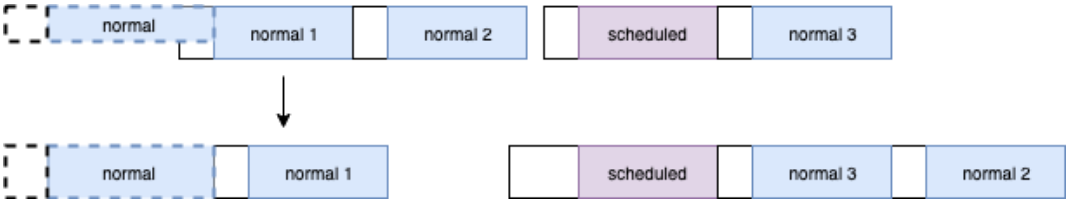

Figure 5.12: Change arrive time - longer order - collision of normal and scheduled order

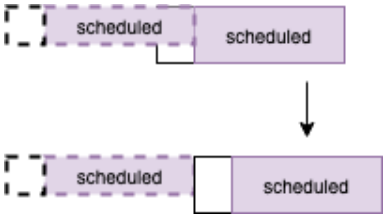

Figure 5.13: Change arrive time - longer order - collision of two scheduled orders

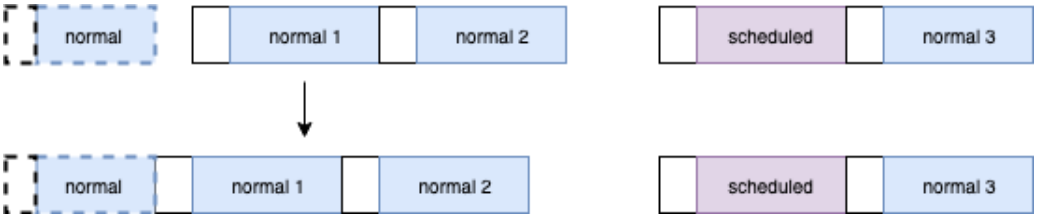

Figure 5.14: Change arrive time - shorter order

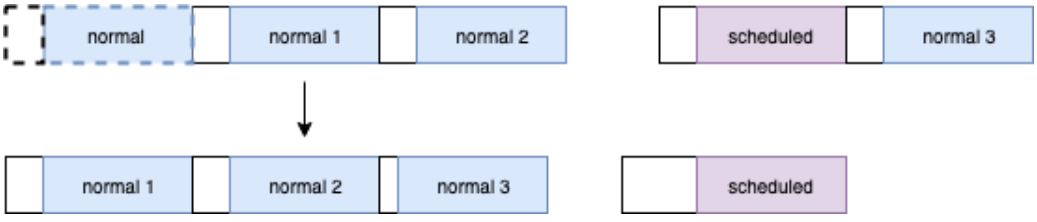

Figure 5.15: Cancel the order

6. Implementation structure

Implementation chapter describes the application architecture and project file structure. Then we focus on implementation details of the most important parts and how are they solved.

Ruby on Rails framework is tightly connected with the Convention over configuration software design paradigm. In short that means that we as a programmer are forced to use Rails conventions otherwise it will not work. This implies that the whole project file structure or class naming conventions are strictly given.

Most of the application is solved using these conventions and tools that the framework gives us. In next section we are going to describe them and the way how we approached to them within our application.

6.1 Project architecture

schema services

database schema

6.2 Application file structure

Ruby on Rails is MVC framework. We suppose, that the reader is familiar with the MVC pattern.

source

Whole Rails application lays in *TaxiBackendApp* folder. In following chapter we are going to describe all of the key files and folders that we are using in our project. All the other files and folders which are not described in this section can be ignored - they are either automatically generated by the framework or we didn't needed that feature of the framework during the development.

These are the key folders and files in our project:

- app
  - controllers
  - helpers
  - lib
  - mailers
  - models
  - policies
  - uploaders
  - validators
  - views
  - workers
- config
  - environments

  - initializers
  - locales
  - database.yml
  - routes.rb
  - sidekiq.yml
- db
  - migrate
  - seeds.rb
  - schema.rb
- lib
- public
- spec
  - factories
  - requests
- Gemfile

6.2.1 app/controllers

In this folder are placed all the controllers we use in the application. All of the controllers in Ruby on Rails must inherit from the *ApplicationController*. Because we are writing API which may have next versions in the future, we decided to add one more layer - *ApiV2Controller* which takes care about the API-specific general tasks such as authorization. This implies that all the controllers for our API are inherited from this controller.

Each controller is separate file in which is just one - the controller class.

Each public method in the controller class is considered as action which can be binded to the router. Default actions are:

- index = display all entities
- show = show specific entity
- create
- update
- destroy

Best practice is to fit with most of the actions on the entities in these names. For action show, update and destroy we suppose that we get entity id from the router.

přidat odkaz na rails guides pro kon-trolery

6.2.2 app/helpers

This folder contains helpers for the application. Helpers are specific functions which are more general for the application and we want to have them separated for better code readability.

In our application was the helpers folder the first choice where to put the calculation of the distance between the two coordinates, because we use it in both controllers and workers and it is quite complex function to have it directly in place.

6.2.3 app/lib

This folder is usually the folder where you place everything which can not be classified to one of the default Ruby on Rails framework predefined folders. Usually it is the code containing custom logic related to the problem the application is solving.

In our case this folder contains whole orders scheduler including the Google Matrix API wrapper.

6.2.4 app/mailers

Ruby on Rails has built-in support for sending emails. For each entity we can create class called Mailer, which can have multiple methods - specific emails and its configurations, based on which the email is later sent. Using this mailer provides us out-of-the box asynchronous queue sending via SMTP - configured in *config/initializers* folder.

The mail template which will be used for the email is specified in the *views/entity_mailer/acti*

In our application we used these mailers for the employees registration and forgotten password mails.

6.2.5 app/models

Models folder contains all the application models. Ruby on Rails provide rich ORM library called ActiveRecord, which is the implementation of the same named design pattern.

přidat citaci na https://en.wikipe - toho týpka coto poj-meno-val

In our application each of the model classes corresponds to one database table. In the model classes we also define relations between the models and validations of the model attributes.

In the models folder there is folder called *concerns*. This folder contains separated logic which is used by more models. In our application we use the concerns for authentication, user roles and notification - all of them are used by the Customer and Employee model, thus it is convenient to have it separated at a separate place.

Another interesting pattern we use a lot in our application are model scopes. Scopes allows us to have defined commonly used filters or queries directly on the model class. For example we have scope *dispatchers* for the *Employee* model, which selects dispatchers only from the set of employees.

6.2.6 app/policies

Policies folder is the place where we have all the permission definitions which uses the Pundit gem. Basic principles how the policy definition works and how it is connected with the rest of the application is described later in the 6.3.1

6.2.7 app/uploaders

Here lies the configuration file for our *Carierwave* uploader used for images of the employees and vehicles.

6.2.8 app/validators

During the creation of the order model we end up with the complicated validator definition. For the better code readability we decide separate that validator to custom file, which lies in this folder.

Even though the orders validator is the only one such complicated that we felt the urge to separate it to its own file, this folder can be used in the future for other complicated validators in the application.

6.2.9 app/views

In this folder is the view part of the application. In the *employee_mailer* folder we can find templates for the emails. Inside the *api/v2* we have the Jbuilder templates for our endpoints. *Index.** is usually for showing collection of specified items, *show.** is for displaying one specific item. Files starting with underscore are called partials. Partials are separated parts of views which are used on more places across the views.

6.2.10 app/workers

In the workers folder lies all the Sidekiq workers. In general we can say that if there is some code running asynchronously in our application, it is in this folder.

6.2.11 config/environments

Here lies the configuration files specific for the development, testing and production environment. Especially we can find there mail and logs configurations.

6.2.12 config/initializers

Ruby on Rails has special folder for initialization and configuration of the modules and classes used in the application. Especially there are initializers for Google Maps API, Sidekiq (asynchronous jobs), Kaminari (pagination), Lograge (logs formatting), CML (our order scheduler) and CORS.

6.2.13 config/locales

Inside this folder we have for each language one YAML file which defines translations for the whole application.dss

6.2.14 config/database.yml

Database YAML file define which database we use on which environment and specifies how to connect to database server and which database use there.

6.2.15 config/routes.rb

Inside routes file we define all our API endpoints. Ruby on Rails has actions index, show, create, update and destroy on each entity by default. It requires for each resource having controller of the same name, where are the actions defined as methods.

6.2.16 config/sidekiq.yml

Inside the sidekiq configuration file we can set thread count which will the job processor run on and also job queue names - in our case mailers and default queue which is processing everything except the mails.

6.2.17 db/migrate

Whole database is created and modified through migrations. Each of these migrations is described in its specific file. Some of them are generated via Rails CLI.

6.2.18 db/seeds.rb

Inside seeds file are defined initial data which are inserted to the database when the *rake db:seed* command is called. In our application we use it for setting up the initial administrator account.

6.2.19 db/schema.rb

This file is automatically generated by the Ruby on Rails from the migrations. It contains whole database schema with the table columns properties and table indexes.

6.2.20 lib

This folder contains in our application only custom Rake task definitions. In our case the generator tasks for generating mock data.

6.2.21 public

Public folder is for the application static assets. Especially this is the place where all the vehicle and employee images are located.

6.2.22 spec/factories

For the mocks generation in our tests we use FactoryBot[1] gem. Factories folder holds definitions of these factories from which mocks are created.

6.2.23 spec/requests

The only tests we have written as part of the thesis are in this folder. Each entity has its own file inside which are the tests structured using the *Context* and *Describe* blocks.

6.2.24 Gemfile

Gemfile specifies library (gem) project dependencies. Each gem record also contains version described based on which the libraries can be automatically updated. This file is similar to the *package.json* file in javascript NPM projects.

6.3 Specific implementation details

6.3.1 Authentication

We created concern *AuthenticableUser* which is included in both Customer and Employees model. The concern takes care of auth token generation and manipulation using *has_secure_token* [2] Rails utility.

ApiV2Controller is the one responsible for checking auth token in headers and setting the current user variable for all the controllers.

SMS workers now just print tokens to logs. When we go to production we just sent this token to some third-party SMS gateway API.

Mailer used for employees token sending is not connected to mail server. All the mails now goes Mailtrap[3], which is a fake SMTP server. In production we must switch to real one. Once we have them, just change the *config/environments/production.rb* config.action_mailer section

6.3.2 Authorization

We use *Pundit* gem to help us with authorization. For each controller in *api/v2* there is one permissions definition file in *politions* folder with the according name. This file contains policy class.

Each method in this policy class is permission definition for the corresponding action in controller. There could also be scope definitions. Their purpose is to return subset of the current entities to which has current user access to. Last type of methods that can appear in these files are custom policy definition methods. These methods check other specific actions needed somewhere in the application and they are called explicitly from views or controllers.

[1] `https://github.com/thoughtbot/factory_bot`

[2] `https://api.rubyonrails.org/classes/ActiveRecord/SecureToken/ClassMethods.html`

[3] `https://mailtrap.io/`

The whole Pundit initialization is in *api/v2/api_v2_controller.rb* where is also defined what the application should do if the request is unauthorized. As we can see, our implementation returns error 403 with 'not authorized' error as specified.

In each controller action we must call the *authorize* method which will automatically checks the permissions for us. If we don't want to authorize the request we must explicitly call *skip_authorization*. In case we don't call it, there will be raised missing authorization exception on such endpoint. This mechanism is there to prevent the situation when programmer forgets to set authorization for the endpoint, which would lead to possible sensitive data exposure.

7. Testing

In the following chapter we describe process of testing the application. Each feature and request has been tested by us after the implementation. Then we published the feature to front-end. Naturally during the implementation they tested the feature again.

In our application we evaluated two critical parts to which we decided adding third layer of testing. First is the authentication and user manipulation part, second is the order scheduler. For these features we followed the *Test-driven development*[1] software development process. When all of the tests for given feature have passed we continued testing the feature such as the other parts of the application.

For the user part of the application we used integration tests. From our experience the process of creating and user activation is almost never revisited during the front-end development thus we loose the the front-end insurance part. Furthermore the user registration and activation process is the most crucial part for the business. When there occurs error during the registration, it is likely that the user will never use our application again.

The reason why we decided to cover the scheduler system was different. First of all we started by analysis of the possible situations. During the development we discovered more and more edge-cases and by fixing them we started to breaking other parts. Also reproduce all the edge cases took a lot of time. Also for the route calculation we use the Google Maps API, which even for the same two points gives different time estimations based on current traffic. In the end we decided to mock the API and write set of tests covering all of the possible situations and desired outputs. Based on these tests we started to implement the scheduler.

7.1 Technologies used for testing

Ruby on Rails has rich support libraries which make the testing easy. As the testing library core we decided to use the *RSpec*[2]. This library has intuitive interface, helpful features and is kind of standard in the Ruby on Rails world.

Next essential part of our testing stack is the *factory_bot* gem[3]. In a short way this library allows us to define multiple factories for each model in the application. When we need an instance of some model, we just use the library's factory and it will return the model instance with the data randomized as described in the factory definition. Also it allows us to specify relations between the factories so that we can easily get an instance of the user with three orders. For the random data generation we use the gem *faker*, which helps us with generating realistic looking random data.

[1] `https://en.wikipedia.org/wiki/Test-driven_development`
[2] `http://rspec.info/`
[3] `https://github.com/thoughtbot/factory_bot`

8. Evaluation

No time to do better orders assignment, custom locations
Things to improve:

- login on more devices at time
- generate login token with each request

Conclusion

In our thesis we have successfully managed to create the back-end part of the application for the taxi companies. We analysed, specified and implemented the whole process of ordering and distributing the orders to the drivers.

Besides the application itself we put together the stack of technologies which allows us to run, distribute and monitor the application.

From the technological point of view our application is ready to be used by the taxi companies in production and improve the services they provide. Unfortunately the pricing change in the API used for the route calculation made the costs per one processed order too high.

8.1 Future improvements

To make the project sustainable from the business point of view we would have to switch from the Google API to other route calculation API. Other possible improvements would be in the order scheduler. First of all we should separate the split the algorithm into the service workers so we would be able to scale the number of processed orders at once easily. Secondly after the few months of operation we could measure the estimated and final durations of the individual order parts and improve the estimations.

Bibliography

Docker Inc. Overview of docker compose, jul 2018a. URL https://docs.docker.com/compose/overview/.

Docker Inc. Get started, part 1: Orientation and setup, jul 2018b. URL https://docs.docker.com/get-started/#docker-concepts.

Petr Havelka. Jak hledat v aplikačním logu, nov 2017. URL https://youtu.be/M8DQ4SRQHko?t=17m33s.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment