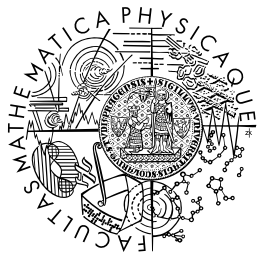

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

late

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

JUST DRAFT! write in sentences, structure it

1. Introduction

We have implemented first version of application in Individual Software Project which is up and running since July 2017. When I refer to the first version of application I mean this.

Do proper introduction, mention somewhere that below:

1.1 Goals

This theses has three main goals

maybe specify more?

- Create application covering order management with respect to existing taxi company processes
- Provide API for front-end(s) which is
  - secure,
  - well documented,
  - easy to use,
  - general enough for later extension to other front-end services
- Put together stack of tools which allows:
  - easy deployment with minimal down time,
  - quick installation,
  - operation system independence,
  - developer-friendly errors & system monitoring,
  - back-ups

1.2 Outline

2. Requirements

In the first part of the chapter we focus on the analysis of creating wholesome solution for taxi company and related business requirements. In the second part we specify requirements from the technical point of view - whether back-end or cooperating front-end. In the end we specify individual parts of back-end application and what they should fullfill.

2.1 Business requirements

JUST DRAFT! write in sentences, structure it

- Handle order system - staff fluctuation is high, train employees is expensive - especially dispatchers, which must have good estimate
- Allow customers to order directly via their phone apps or other sources - saving costs of dispatching. Because it is very easy to order via phone call make it as easy as possible
- Customers must have overview how much their order will approximetly cost and must be able to select their driver. Also they must have approximate estimate when their taxi will come.
- Customers should be informed about their ordered driver status - where he is and when them will come
- better customer service - remember name, favorite location, other things
- security - strong competition, already few cases of hacking and data stealing
- some of the things written tailored for our local taxi company but should be easily rewritten and reused for some other company with differnet requirements and priorities
- predicted data size for the project for now: 10 drivers - 7 online, 7 cars, 10 dispatchers - 2 online, 6000 customers, 800 orders per week
- authorization
- two main types of orders - scheduled on time and normal. Our high priority is to be there for scheduled precisely on time.
- handling frauds

2.2 Technical requirements

- Separate back-end and frontend - allows more customization on frontend which can and will change more frequently totally independently on back-end. On frontend - where are apps and PWAs and orders through FB messenger for example.
- HTTPS

- Frontend-developer friendly instalation and independency
- Able to install easily on new server with possibility to scale - at least preparation
- Errors logging and alerting

3. Specification

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. 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
- source - whether it was created by dispatcher or directly by customer via front-end application

Also with order we want to know these information about times. 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.)

With order users can manipulate with these actions:

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

Because the order process Illustration displaying all the order statuses, actions what we can do with the order and actions which must our system do.

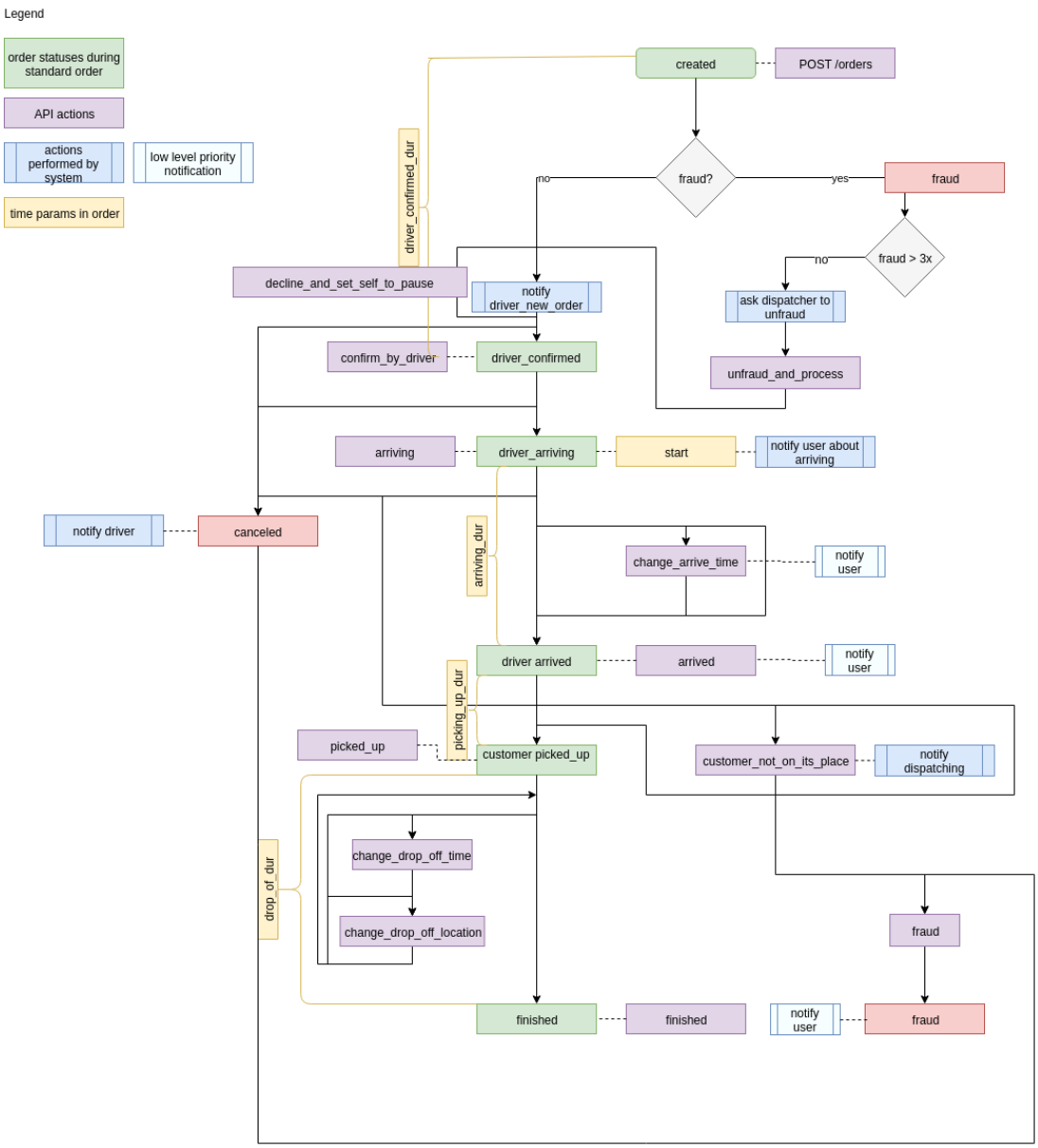

Figure 3.1: Order process scheme

3.4.1 create

3.4.2 show allspecific

3.4.3 my orders for dispatcher

3.4.4 driver arrivals

3.4.5 confirm by driver

3.4.6 arriving

3.4.7 change arrive time

3.4.8 arrived

3.4.9 customer not on its place

3.4.10 picked up

3.4.11 change drop off time

3.4.12 change drop off location

3.4.13 finished

3.4.14 fraud

3.4.15 unfraud and process

3.4.16 cancel

3.5 Notifications

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

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

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.

[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. Analysis

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 to our project more secure but also much more complex way of authenticating. Once client gets the token from the login endpoint, the token is generated and returned with each following request. Using this approach we have to solve the batch request problem. Imagine that client sends for example three requests at once to the server. Of course all the three requests must have same authentication token - because client doesn't have the new token until the response comes. Thus server must accept all those three requests as authenticated, but also they must agree with the front-end 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 changed. The front-end library was supposed to implement that, but after the three weeks of trying and founding several bugs in the library - even without successful login they 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].

Sending verifying tokens

We must send tokens via emails and SMS during the whole authentication process. Because these are actions that are not instant (API request, SMTP request) and because 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 have instant response without blocking webserver threads during account creation/recovery and have ability to automatically resend the message in case of the third party or network failure.

5.1.2 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 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

[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`

them written as simply as possible, this was the main reason why we chose Pundit over Cancancan.

5.1.3 Pagination

paginating lists - why, where

5.1.4 Request parameters security

Permiting

5.1.5 Rendering views

5.1.6 Images

- too difficult for front-end to implement and handle api multipart requests - limitation only one image for entity =¿ but enough for us - employee and vehicle images

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.

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[6] 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 was how to handle 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

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

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:
  - 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 Orders

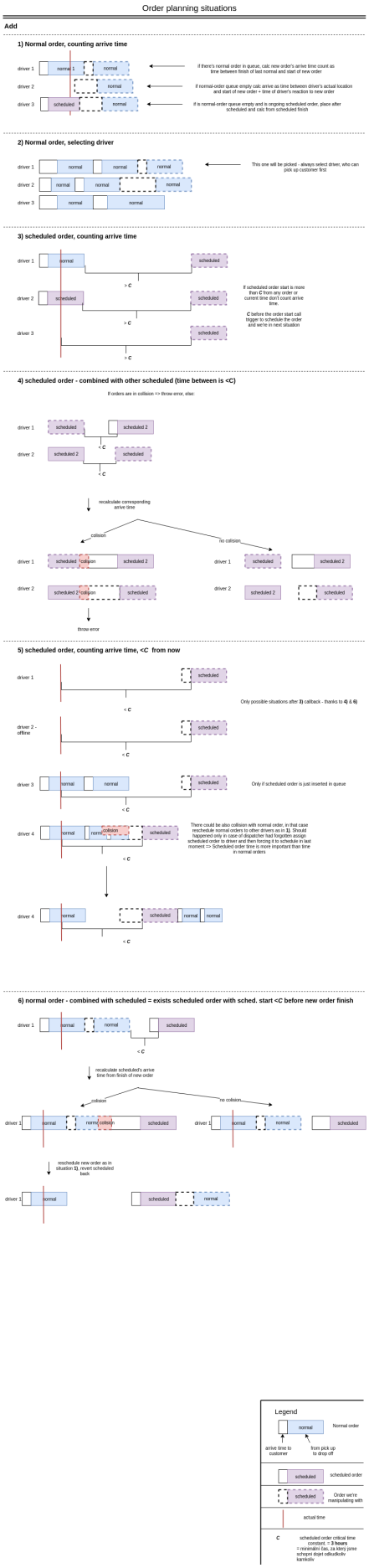

Figure 5.2: Order planning situations

6. Implementation

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. If you are already familiar with Ruby on Rails framework, you can freely skip next section and explore the source code.

6.1 General architecture

describe worker and mailer directory.

6.2 Specific implementation details

6.2.1 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* [1] 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[2], 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

6.2.2 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.

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

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

These methods check other specific actions needed somewhere in the application and they are called explicitly from views or controllers.

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.

6.3 Customers

6.3.1 Create

We ended with three endpoints: create, confirm and resend confirmation.

6.3.2 Customer favourite location

6.4 Employees

how email links are stored in env variables

7. Testing

- unit tests covering
- manual testing
- testing by front-end developers

8. Evaluation

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

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

Conclusion

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