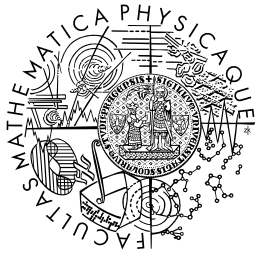

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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

I would like to thank my supervisor for his kind approach and useful development advices, tips and comments during the whole project. My special thanks belongs to my wife and my whole family for the patient support and encouragement during the whole studies. I would also like to thank the Taxi Ali Mladá Boleslav company for providing us with the taxi company business insights and their consultations.

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. In this thesis, we create a back-end part of an application for taxi companies in smaller towns. This application provides an interface for creating and managing orders, employees, customers, and vehicles. As part of the system, we constructed order scheduler which calculates the order duration, estimated time of arrival and distributes the order between available drivers automatically. Overall the application increases the taxi company's efficiency and offers users a more comfortable experience.

Keywords: Taxi Ruby on Rails Web application

Contents

1. Introduction

The field of personal transportation has been revolutionized in the past few years. People in metropolises got used to ordering the taxi via a 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 order. Also, the service during the order process has evolved a lot. People are used to seeing 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, the situation in smaller cities is very different. None of the big companies operates in 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 dissatisfied. Providing this new way of ordering would have a massive impact on the company's 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 smaller towns. We cooperated with the company from Mladá Boleslav, which provided us with 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 a taxi company is very stressful and low paid job. The main working hours are at night. These conditions lead to high staff fluctuation. A dispatcher is responsible for distributing orders between drivers and estimating arrival times. When the dispatchers are changed so often, their estimations tend to be highly inaccurate which leads to fatal failures and very unsatisfied customers. Having the algorithm for distributing orders between drivers and estimating arrival times leads to more stable estimations which can be further improved using the gathered data in the future. This reduces the skills the dispatcher needs to have for the job - now its job is just to communicate with the customer and enter the orders into the system.

The second advantage of having such a 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 a second dispatcher.

1.2 Existing solutions

As we mentioned before, there are already existing solutions to this problem. The most known companies operating in the Czech Republic are Uber, Taxify, and Liftago. They all provide the same feature - allowing the user to order the taxi via 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. They prefer to give these orders to their most experienced and loyal drivers.

1.3 Goals

Our goal is to create an application which will automate the ordering system for the taxi company.

The application will contain an authentication system using which the drivers, dispatchers, customers, and administrators can log in and be identified. The application will also contain basic authorization system so that for each action (e.g. creating driver) we specify authorized entities (e.g. administrators) who are allowed to perform it.

The ordering system allows dispatchers and customers to create the order. It connects the customer with driver and leads them both through the whole process of order. From the order creation to the desired destination arrival.

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

We design the REST API through which the front-end applications will communicate with our application. The front-end applications are not part of this thesis. They have been implemented by Patrícia Březinová in her bachelor thesis.

1.4 Outline

In the beginning we introduce the problem the thesis is dealing with and set the goals we want to achieve. We describe the current situation in the taxi services field and uncover our motivation to build the application.

In Chapter 2 we formalize the whole ordering process of the company. We specify all the entities and actions that our application uses and provides.

Chapter 3 describes the tools we decided to use for our application and why.

In Chapter 4 we describe the problems that we deal with in the application. We describe in detail how we solve them and why we have chosen to solve them that way.

Chapter 5 reveals the application implementation details. We explain the application architecture and give the reader insights into the project structure.

In chapter 6 we describe how to get access to the REST API documentation we provide.

Chapter 7 describes how we tested the whole application and which technologies we used for the testing.

In chapter 8 we evaluate the goals we set in the introduction and summarize what we have accomplished.

In conclusion we evaluate the project as a whole and suggest a direction the project could be eventually improved in.

2. Analysis

After several consultations with the taxi company from the Mladá Boleslav we found out that in order to create such ordering system, our application must provide an interface to work with these five entities: Customers, Employees, Vehicles, Orders, and Notifications.

In this chapter, we define each of these entities. We describe the data we want to store about them, the actions that can be performed with them and who is authorized to perform each of these actions. The overall goal of this chapter is to formalize the taxi company ordering process.

2.1 Customers

Customers are uniquely identified by telephone number. We also want to store about each of them this 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 a specific customer

2.1.1 Operation create and confirm

There are two ways of how the customer can be created. It is either directly through registration or indirectly by creating a 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 a provided password. The application must verify the given telephone number.

An indirectly registered user is created during the creation of a new order using a telephone number, which doesn't belong to any existing customer. This customer type is just an envelope with the purpose of information tracking and

statistics - mostly for better customer support. This account type cannot be used for authentication. Indirectly registered user can be later registered directly with no difference to the normal direct registration.

2.1.2 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.

2.1.3 Operation destroy

Destroying the customer can be invoked either by the customer itself or via the administrator. Orders made by that customer are not deleted - we just remove the information about the customer from the order.

2.1.4 Operation password recovery

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

2.1.5 Operation login and logout

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

2.1.6 Operation list favourite locations

Our application must provide a list of customer favourite places. In the front-end applications is this operation called when the customer chooses its pick-up and drop-off location.

These places should be ordered with respect to the given location (current customer location or the selected pick-up location when choosing drop-off). It should also take into account whether the customer chooses the pick-up or drop-off location. These recommendations should be based on the customer's order history and respect the start-finish relation of the orders. The most important are the first five places returned, so the place that the customer is most likely to choose based on current conditions should be amongst them.

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

Only the user itself or the employees are able to see the favourite locations list.

2.1.7 Operation list all customers and show specific customer

Our API must provide information about 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 the administrator only. The list of the customers is one of the most valuable taxi company's assets so we don't want to provide it for the staff. Of course that 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.

2.2 Employees

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

- Email
- Name
- Photograph

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

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

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

2.2.1 Operation create and confirm

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

2.2.2 Operation update

An employee can update its password, name and image. Besides these fields, an administrator can change the employee roles also.

2.2.3 Operation destroy

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

2.2.4 Operation recover password

Recovering the forgotten password is exactly the same as in the customers' case - except the reset password token is hidden in a link sent via email.

2.2.5 Operation login and logout

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

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

2.2.7 Shifts

Employees could be in three states:

- 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 a flat tire and cannot continue. Employees will be instructed not to do so for better customer experience.

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

2.2.8 Driver locations

Each driver sends since the shift start until the shift end its location in regular intervals. Driver's current location can be set only by the driver itself. See the driver's last available location can anyone, so the front-end can display the current location of arriving driver even for an anonymous customer.

2.2.9 Driver order queues

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

2.3 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 this information:

- name
- internal taxi company vehicle number 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

2.3.1 Operation create, update and destroy

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

2.3.2 Operation show all and specific vehicle

Administrators can see all the vehicles, others can see only the active ones. Operations reveal all the attributes besides the internal vehicle number and the availability to anyone. Employees can see all of the attributes.

2.4 Orders

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

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

Order is created from two sources - dispatchers and by customers directly.

About each order we would like to keep this 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 cannot be changed (is explicitly chosen)
- VIP (just internal flag for the taxi company)
- flight number - in case the order is to/from the airport
- customer
- assigned dispatcher
- date and time the customer wants the taxi to arrive at the pick-up location (scheduled pick-up)
- source - whether it was created by dispatcher or directly by customer via front-end application

Also with the order we want to track these time parameters. In parenthesis is described how the time fields will be referenced in the whole thesis and the application:

- created time
- application estimate when the driver departs for a 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 complex, we decided to sum it up in one illustration. This scheme displays all the order states (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).

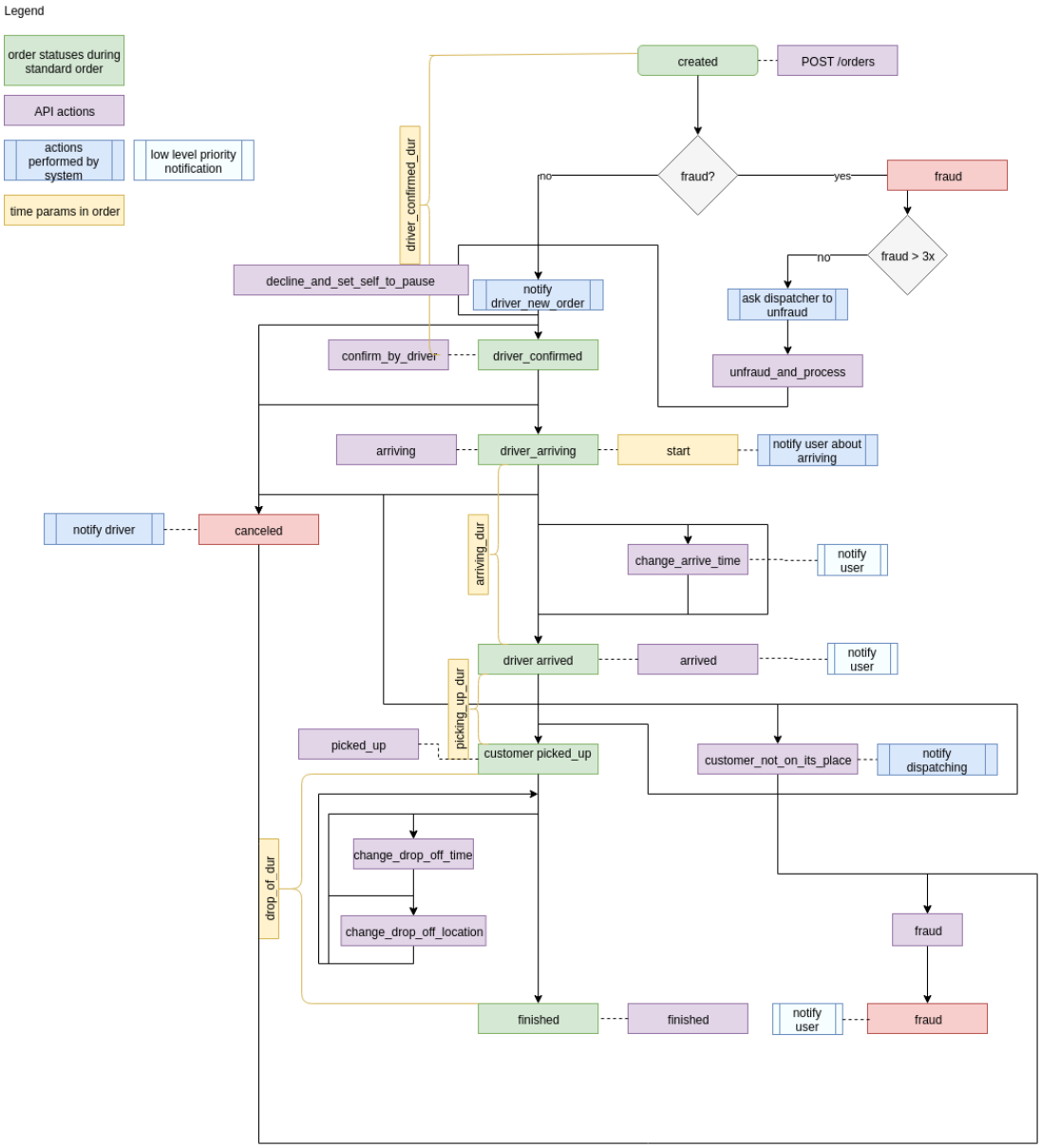

Figure 2.1: Order process scheme

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

2.4.1 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.

A customer whose order it is can also see the coordinates and the addresses of start and finish, passenger count, note, contact telephone, estimated price, whether the 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 the assigned dispatcher and all the tracked times, estimations and original estimations.

Orders support basic filtering. We can get orders which are created since and until a specified time passed as parameter. We can retrieve also only scheduled orders. Orders can be filtered based on their status - we can set multiple statuses in which all the returned orders are.

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

In a request for showing all the orders, we show the total order count for the specified query.

2.4.2 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 for that location. This information then dispatcher tells the customer in case the order is via dispatching. In the other case, the customer can see it directly in the front-end application.

This estimation doesn't have to be as precise as the estimation after the order is created.

Same as in order creation it takes the driver parameter for the case that the customer wants the specific driver.

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.

2.4.3 Create order

Order can be created in two ways. Either it is via dispatcher - customer calls the taxi company and the dispatcher will make the order or directly by customer via an app.

If the customer was marked as a fraud before, a new order is automatically marked as fraud one, thus it does not continue in the process and the creator is informed about that. If the fraud orders count in customer's history is less than three, one of the dispatchers on shift gets the notification whether this customer should be forgiven and the order can be 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. There is an exception for the scheduled orders - they are assigned to drivers exclusively by the dispatchers.

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

After this action, order status is *created*.

2.4.4 Defraud and process

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

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

2.4.5 Confirm by driver

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

This action changes the order status to *driver_confirmed*

2.4.6 Refuse by driver

A driver can also refuse an assigned order. We suppose that this will happen only in emergency cases. If the driver refuses the order, it is removed from its queue and passed on to the next available driver. If there's no one available the order is marked as cancelled.

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

2.4.7 Arriving

A 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 the pick-up location. At this moment it should very precise estimation - the driver should be slowed down by the traffic only which is included in the Google Maps API. Thus at this moment, we send the SMS to the customer with the estimated time arrival.

In order to call this action the order status must be *driver_confirmed* and it can be called by the driver assigned to order only.

2.4.8 Change arrive time

A driver can correct the estimated arrival time based on its experience or unexpected complications during the journey.

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

2.4.9 Arrived

This request is sent when a driver arrives at the pick-up place. This changes the order status to *arrived* and updates following times estimations because at this moment we know the real time of arrival.

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

2.4.10 Customer not on its place

In case the driver is at the pick-up location and cannot get in touch with the customer, this action is called. It sends the notification to the dispatching which takes care of the situation and communicates the problem between the driver and the customer. This situation can be resolved in two ways only. Either the customer is found and picked up, or it is not and the order is marked as fraud.

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

2.4.11 Picked up

In a point when the driver picks up the customer and is ready to ride to drop-off location this request is 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 the order is.

2.4.12 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.

2.4.13 Finished

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

2.4.14 Fraud

Order can be marked as fraud from the *driver_arrived* and *cancelled* order statuses. Mark order as fraud can the driver assigned to the order only. Marking the order as fraud resets the customer's fraud counter back to overall fraud orders count increased by one.

2.4.15 Cancel

Order can be cancelled by the driver assigned to order, dispatcher, or the customer whose order it is. In case the customer is anonymous it can be cancelled by anyone

- because we cannot distinguish separate anonymous users.

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

2.4.16 My orders for dispatcher

Each order is assigned to a dispatcher. This dispatcher then takes care of the customers and in case of a problem must be able to react and handle it with the customer. For such purpose, there is this endpoint. A dispatcher can see all its orders with all important details such as the current 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.

2.5 Notifications

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

- driver has a 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 with telephone number. *New order from fraud customer* contains order and customer id.

Notification system is passive = front-end applications call our *list notifications* action in regular intervals. User can mark the notification as resolved.

2.5.1 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 the unresolved notifications.

2.5.2 Operation mark as resolved

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

3. Technical Analysis

In this chapter, we want to take a look at 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. Primarily what frameworks and tools we use and why we have made such a decision. In the second part, we focus on the whole server stack and how we manage to run all the services on it.

3.1 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.

3.1.1 Ruby on Rails

There are many frameworks which could be this type of application written in equally well. For example the 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 the clarity to project and most of the times it gives you a good way to solve your problem without reinventing the wheel. The biggest advantage of using this approach is that a new programmer understands the project architecture quickly because all the Rails application share the same architecture skeleton.
- 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 the ecosystem

[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/`

Disadvantages:

- it is more difficult to set it up than PHP
- impossible to use standard web hosting
- small base of programmers knowing Ruby

We tried to low the disadvantages as possible. We reduced the set-up difficulty by using the Docker. Using the web hosting would be limiting for our application so we would have to use own virtual server anyway.

3.1.2 PostgreSQL

We decided to use PostgreSQL because it is open source, and unlike MySQL, it supports transactions, natively storing JSON, arrays and it has many plugins (e.g. plugin for storing geodata). Besides the transactions 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), the choice of the database is not so critical - we can later migrate to the other database engine easily.

3.1.3 Sidekiq

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

3.2 Tools stack

As our REST API documentation tool we use Apiary. The tools we decided to use for easier installation and deployment is Docker with Docker-compose. As our reverse-proxy and HTTPS certificates manager we decided to use *jwilder's ngnix-proxy*[7] image set. As an error catcher and alerter for all of our running apps, we decided to use *Errbit*[8]. For the server monitoring and alerting in the future we decided to use *uschtwill's docker_monitoring_logging_alerting*[9] image set.

We are aware that the server stack we ended up with is a little bit overkilled for our use case. In our defence, we learned a lot during these services set up. These services also provide lots of handy features.

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

[7] `https://github.com/jwilder/nginx-proxy`

[8] `https://github.com/errbit/errbit`

[9] `https://github.com/uschtwill/docker_monitoring_logging_alerting`

3.2.1 Apiary

We needed a 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 the Czech Republic, thus we knew it before and knew that it fulfils all of our requirements. It even allows 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.

3.2.2 Docker and Docker-compose

Our goal was that each frontend developer would have its own local backend application instance against which they could develop. To achieve it, we had created a tutorial on how to install Ruby, Rails, PostgreSQL and all the software 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 the stack working on Windows which was a critical 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 allowed us to set up a container for each service (application, sidekiq, database, etc.) and run them smoothly on any operating system using only one command - *docker-compose up* - without any significant performance decrease[10].

3.3 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 the 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. Also adding a new site is just the matter of starting a new container with three environment variables.

[10] from our observations during development

3.4 Errbit

Our goal was to have a service where we could see all of our apps' unexpected failures. This service should notify us whenever this failure occurs. We should be also able to get at least basic info when, where and on what environment that failure occurred.

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

3.5 Monitoring

Because our server stack is composed of more 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 the current system status and health from a web browser. During the research, we have found a 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 inspired by Peter Havelka on Hradec Kralove Barcamp, who said, that they are using the exact 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]

4. Solution design

In this chapter, we would like to explain the thinking process before and during the implementation of the back-end application. We explain the problems that we have to solve in the implementation part. We reveal the possible ways to solve the problems and describe how we solved them in the end.

4.1 Authentication

In our application, we had to deal with authentication for customers and employees. There are many ways how users can be authenticated using the REST API. We ended up with token authentication - in each request frontend applications must set the Authorization header with the authentication token as a value. Front-end application can retrieve the token in exchange for correct credentials.

Another part of the authentication we have to consider is at least the basic security during the manipulation with authentication data and flawless verification token sending.

Token authentication details

We decided to implement the token authentication on our own, using only Rails helpers for generating and verifying secure tokens. As you can see in 4.1, the token is generated during the login action and then returned in a body response. In order to be considered authenticated, all the following requests must have this token present in the headers .

Our application store just one token per user. That implies the user can be logged in from one device at a time only. Having more valid token for users would let into several problems with the invalidation in case of logging out.

In Ruby on Rails there exists a very good gem for authentication called *devise_token_auth* [1]. This library is always the first choice if a Ruby on Rails developer implements the authentication system. However, we decided not to use it in the end. In the following paragraphs, we explain why.

The first reason for not using the gem is that it uses an email address as the default authentication field. For the customers we wanted to use the telephone number as the main identifier, which would lead us to rewrite most of the library's controllers and models anyway.

The second reason was the complexity of the whole authentication process which the gem forced us to use. This library would provide a more secure but also much more complex way of authenticating in 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 arises the batch request problem. Imagine that the front-end application sends three requests to the server at once. Of course that all of the three requests must have the same authentication token - the front-end doesn't have the new token until the response. This implies the server must accept all of those three requests as authenticated. Besides that the back-end and front-end must agree on which of the responses has the new correct

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

Figure 4.1: Auth token scheme

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 - without even successful login - we gave up using it.

This implies that 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 from scratch.

We came to the conclusion that the complexity which would bring this library to our application is not worth the better security and stability we would gain from using it. Forcing the HTTPS protocol for the communication between the server and front-end applications makes the token theft more difficult in this case. Also, the only really sensitive accounts for identity theft are the employees' ones. Furtunatelly we are in personal contact with them regularly, so we get to know about the potential account misuse or hack quickly. Also, we can provide a more secure and better solution later.

Basic security

As we mentioned in the 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.

As a result of using the Ruby on Rails tools correctly we also excerpt the passwords from all of the application logs, and we are not vulnerable to SQL Injection[3] attack.

Verification tokens sending process

We send the verification tokens via emails and SMS during the account manipulations. 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 via Sidekiq workers. In exchange for some time spent on the configuration we don't block web server threads during account creation/recovery token sending and we also have the ability of resending the messages automatically in case of the third party or network failure.

4.2 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 the version control system. Those files are then loaded by the Docker, which sets them as environment variables. From these variables we read this sensitive information in the whole application.

4.3 Authorization

Almost every endpoint has its own rules of who and in what circumstances can do this 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 those types.

In each request we solve the condition whether the current user is able to execute the requested action or not. Having all these conditions directly in controllers would make the controllers difficult to read. Also, these conditions could (and did) have changed during the development iterations. This led us to have the authorization conditions separated in the different part of the application.

We wanted to avoid reinventing the wheel so we have chosen to use the *Pundit* [4] gem.

Another option came across during the research was the *Cancancan* [5] gem. They both have a long maintenance history, they are actively developed in the time of writing and they satisfy all of our requirements for the authorization. *Cancancan* is better suited for the applications with complicated views because it provides more helpers for checking the authorization in them. Its architecture is also more general thus it is better optimized for more complicated permission

[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`

management. That results in slightly more complex permission definitions. On the other side, the *Pundit* is a more light-weight 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*. .

4.4 Pagination

As the data grows some of the requests (e.g. order index) would return thousands of items. This would be a huge waste of resources, so we have to limit the count of returned items per request and enable pagination. For such purpose, there exists a gem *Kaminari*[6] which adds methods for quick filtering and retrieving data in our models. 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.

4.5 Rendering views

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

At the beginning of the implementation we thought that we don't have to have the view part separated from the controller. For 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 obscured the controllers.

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

4.6 Images

Our employees and vehicles contain images. We had to figure out how to get the image from the frontend application via the REST API to our backend application.

We ended up with two options on how to implement it. First is the multipart HTML requests[8]. The second option is to encode the image into Base64 on frontend and then send it in a request body as JSON field. This field is then decoded on the server side and saved.

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

[6] `https://github.com/kaminari/kaminari`
[7] `https://github.com/rails/jbuilder`
[8] `https://tools.ietf.org/html/rfc7578`

transfer as much data this way as in the multipart request option. However, we have only one image for each employee and vehicle thus this solution is sufficient.

For the back-end side we decided to use the *Carrierwave::Base64* [9] gem. Besides the out-of-the-box Base64 encoding of a specified attribute which we need, it provides the simple API for the storage settings such as file names. Also in case of switching our storage to some cloud storage provider such as the S3 by Amazon[10] in the future, it would be just the matter of changing few lines in the configuration.

4.7 Customers

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

- In case of emergency during the order process we must be able to contact the customer immediately, so we need the customer's phone anyway.
- Customers are going to register and order taxis from their phones most of the time.
- Not everyone has direct access to the mailbox from its the phone - unlike the SMS which has every phone built-in.

4.7.1 Create and confirm

Besides the authentication problems described in before, we have to deal with the telephone verification.

We decided to use verification via the SMS code.

A 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 a registration token. This token is valid for 5 minutes, and the customer can ask for resending. Resending will invalidate the last token, generate a new one and send it. Confirm is made with provided token and the telephone number.

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

4.7.2 Password recovery

The 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 the first endpoint - password recovery - sends in exchange for the telephone number SMS to that telephone number with password recovery token.

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

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

Then the customer can send a 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. These parameters we just picked are similar to the other services on the internet. Of course we are able to change them later easily if we discover that it's not suitable.

We have noticed that the *password_recovery* endpoint must - except the bad request format - always return the success status. Especially we can not let the client know that the account with such telephone number was not found and neither that the SMS was not sent successfully. If we would return an error code in such a situation, an attacker could get all the telephone numbers of all of our customers this way.

We are aware that the SMS may not be the most secure way of verifying users[11] 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 scenario is that the attacker creates an order on behalf of the victim and thus that order will be a fraud. However, the fraud orders occur a few times a week anyway.

4.7.3 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.

The first 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 acquires one of the following values:

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

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 a 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 later recommend it just once. On the other side, the tolerance must be small enough to not to join together two different places. We thought about using some kind of modified clustering algorithm but we ended up with a conclusion that this would be an overkill in our case . We came to a conclusion that for our use case is having some distance tolerance constant defined enough. Then all of the places within this distance are considered as one place. We set this constant to 50 meters because it seemed like a good compromise between the inaccurate low-end GPS sensors precision and the distance between two different places. Of course, we are able to change this constant in the future in case we discover that it is too small or big.

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

The second problem was how to filter and return the list of places fast. For each user, we have an index of visited places with the information based on which our algorithm recommends. In this index we have the following information about each place:

- coordinates
- list of items containing for each place occurrence in orders:
  - a 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 a request, we go through the index place by place and count its weight as a sum of the weights where the place occurs multiplied by a constant if start/finish fits the desired index direction. The index is also prepared for the places recommendations based on the order time (e.g. in the evening we go to the 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. We are aware of the fact that it could be more effective and recommendation could be more sophisticated. However, we came to this algorithm during the development and it satisfied all our desired requirements so we kept it.

4.8 Order scheduler

The goal of order scheduler in our application is to distribute the new orders between the drivers. Besides that, the system accepts other actions for order manipulation. These actions are then reflected to the corresponding driver's queue and to all of 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 pick-up and the drop-off location coordinates. Based on this information we have to calculate the distance and the duration of the order. Each order consists of the two parts we have to calculate. The first part is from the pick-up location to the drop-off location. This part is fixed for each of the orders. The 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* [12] which allows us to get the distances between all the drivers and the order pick-up location in a single request. Also choosing this API allows us to use the existing Ruby Gem *googlemaps-services*[13]. This library provides the Ruby

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

interface for the communication with that API. This API can also calculate the distance and duration with respect to the current traffic.

The 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 automatically but the dispatchers do that manually. This implies that in our system could appear order collisions which must the system handle correctly. We have to take in to account the high cost of each route calculation too.

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

- Add
- Change arrive time
- Cancel
- Change driver

In the following subsections we describe possible situations that can occur in each of these actions, how the planning system should handle them and how it should respond to them. We display the situations using the diagrams described in the legend 4.2.

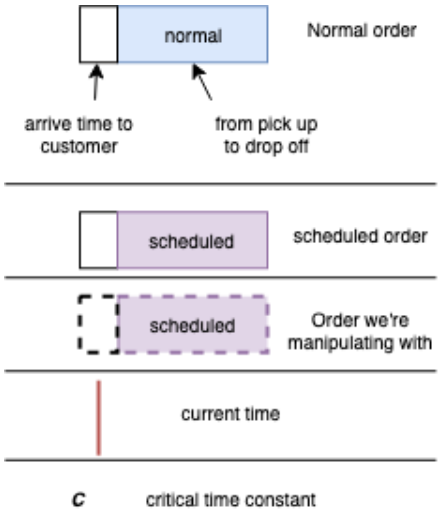

Figure 4.2: Order planning scheme legend

4.8.1 Add order

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

Order can be assigned to either any of the available drivers or to a specific driver. By available driver, we mean the driver who is on shift and is not in the pause state. From this set of drivers, we choose the driver who will be assigned to the order. In case that there is no driver suitable scheduler raises an error.

As mentioned before, the scheduler must ensure that if the scheduled order is assigned to the driver, the driver is able to get there right on a specified pick-up time. This reveals the problem - how to calculate the time when the driver should start arriving towards the customer (*start_est time*)? To calculate it we must calculate the arriving duration between the driver location and the pick-up location. The problem is that the expected driver location changes with each assigned order.

First naive solution led us to count the start time when we add the order to the queue. In this point we know the finish location of order before, so we could count arrive time from that point. The problem is that the scheduled order can (and probably will) start a week or two from now. In order to persist the calculated start time, we have two options. First one is to forbid adding new orders before the scheduled order. This is obviously unacceptable. The second one is that during the insertion of some other order in between the last order in the queue and our scheduled order, we calculate two start times - the newly inserted one and the scheduled one 4.3.

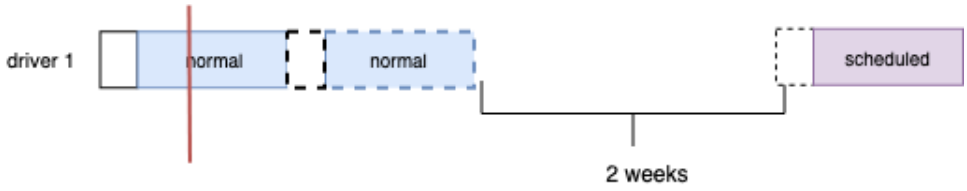

Figure 4.3: Scheduled order - naive arrive time calculating approach

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

We observed that there must be some point when the start time of the scheduled order must be calculated. If we calculate it too early, we raise the costs per order. If we calculate it too late, the driver may not be able to get to the pick-up location on time.

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

In our scheduling system then holds this fact: For each scheduled order in a queue, which has pick-up time less than C from the current time or the finish time of order before, it holds that its start time is calculated. This time is then updated with each relevant change in the queue. If the distance between two orders in a queue is larger than C the start time is equal to pick-up time.

The whole process of calculating the arrival and start time led us to split the problem of adding the order to queue into seven parts:

- Normal order - calculating the arrival time
- Normal order - selecting the driver
- Scheduled order - calculating the start time

- Scheduled order combined with other scheduled order
- Scheduled order combined with normal order
- Normal order combined with scheduled order
- Scheduled order - scheduling at critical time callback

Normal order - calculating the arrival time

First, we focus on how to calculate the normal order's arrival time. As we can see in 4.4 there are three states in which the queue can be.

As in the situation *1a)* if the last order in queue is a normal order, calculate the arrival time as the sum of the last order finish time and the duration between the orders' finish and start location.

In case that the queue is empty as in *1b)* calculate the arrival time as a current time plus the driver's average response time to order plus the duration between the last driver's known location and order pick-up place.

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

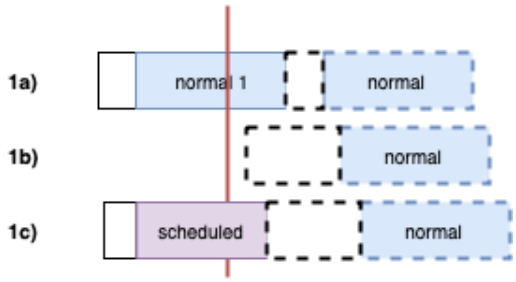

Figure 4.4: Normal order - calculating the arrival time

Normal order - selecting the driver

As described in specification normal order must be always assigned to the driver who can pick up the customer first. This rule demonstrates the example 4.5. The order will be assigned to the *driver 1* even though the *driver 2* is available first.

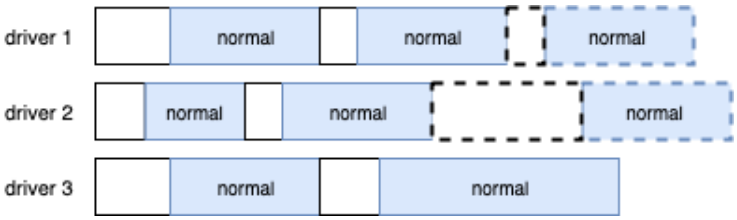

Figure 4.5: Normal order - counting arrive

Scheduled order - calculating the start time

If we add a scheduled order to the driver's queue in which there is no other order within the critical time constant, we do not calculate the arriving duration of that order4.6. We are allowed to do so because the critical time constant C ensures that even if the queue stayed in the same state, the arriving duration between the last driver's location to the order start will be less than C.

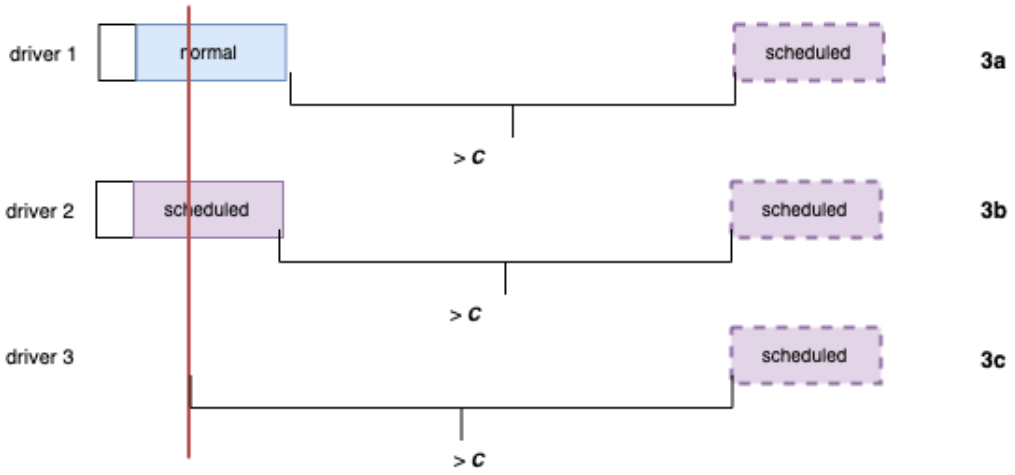

Figure 4.6: Scheduled order - counting arrive time

Scheduled order - combined with other scheduled order

In case that the scheduled order has other scheduled order within the C from the calculated pick-up or finish time, we have to calculate corresponding order start times 4.7. After the calculation, we may encounter the orders collision. In such a case, the order cannot be added to the queue and the order scheduler must throw an error.

Scheduled order combined with normal order

When the expected finish time of last normal order in queue is within the C from the inserted order's scheduled pick-up time, we must calculate the arriving duration. In case that the the scheduled order is in collision with some normal order, scheduler must throw an error 4.8.

Normal order combined with scheduled order

Adding a new normal order to queue could break the fact that each of the scheduled orders in our system has the arriving duration calculated 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 a collision. In case we encounter the collision, we start the process of adding the order again. In this second try we consider the scheduled order as a last order in this queue so in case of choosing this queue again, the normal order

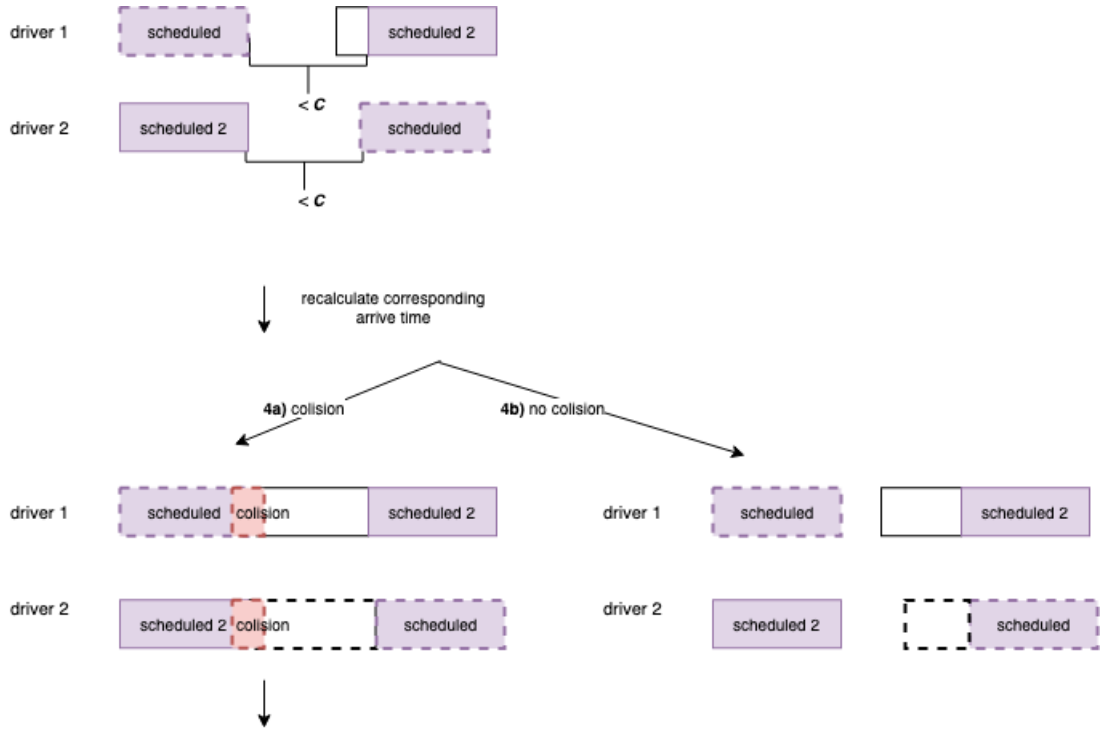

Figure 4.7: Scheduled order - combined with other scheduled

will be inserted after it. Also the arrive time of the colliding scheduled order must be set back to the original value. 4.9.

Scheduled order - scheduling at critical time callback

As we know, scheduled orders do not have the arriving duration calculated only when there is no order in queue with a finish time within the C from the pick-up time or the pick-up time is not within the C from 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 cover with a callback which is called exactly in that point. If in this point the order does not have the start time calculated, we calculate it from the last known driver's location and the order start location. 4.10.

4.8.2 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.

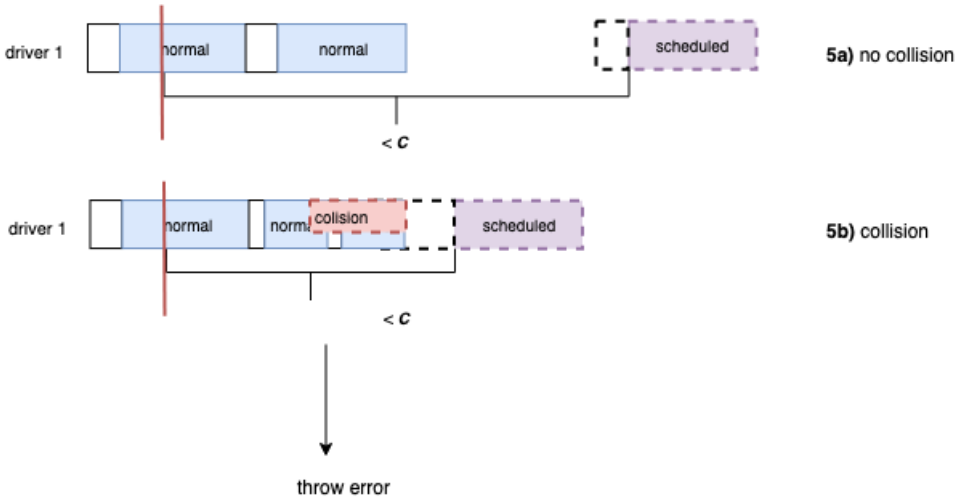

Figure 4.8: Scheduled order - combined with normal order

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 accordingly4.11.

During this change there can occur a 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 4.12.

If the collision causes other scheduled or an ongoing order, we must shift the start of the scheduled order 4.13.

Shorter order

When the time change makes the order shorter, we just simply change the start time estimations accordingly for all of the normal orders after the changed one until the first scheduled order in queue. 4.14.

4.8.3 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* 4.15.

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

4.8.4 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.

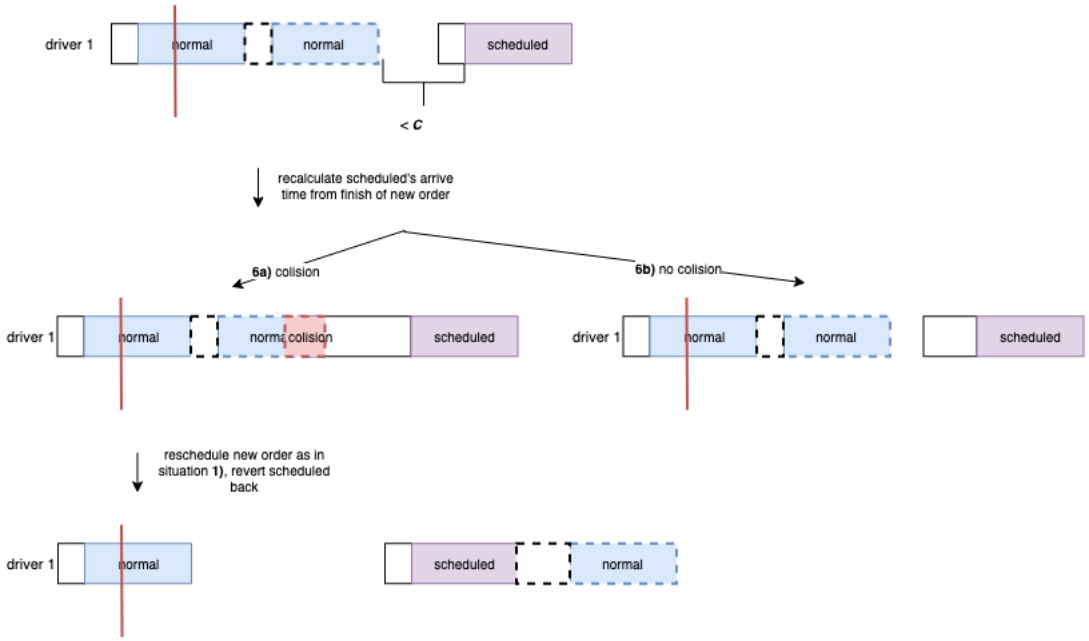

Figure 4.9: Normal order - combined with scheduled order

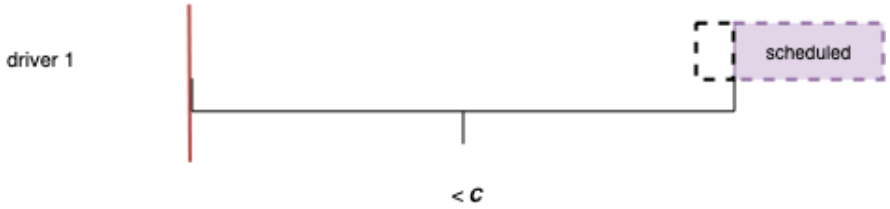

Figure 4.10: Scheduled order - critical time callback

4.9 Orders

For our application it 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 with improving the scheduling system. We have found the *paper_trail* gem[14] which fulfils our needs and is easy to use and setup.

Second problem we have to deal with in the orders was how to store the time information we are keeping the track of. In one half of the application we have to know and modify the time durations (arrive, picking up) and in the other half we have to know exact times (start, arrival 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 the durations in seconds for the rest. Thanks to this approach we can easily manipulate the orders in scheduler. Downside of this approach is that we can not easily filter and sort the orders based on those calculated time attributes.

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

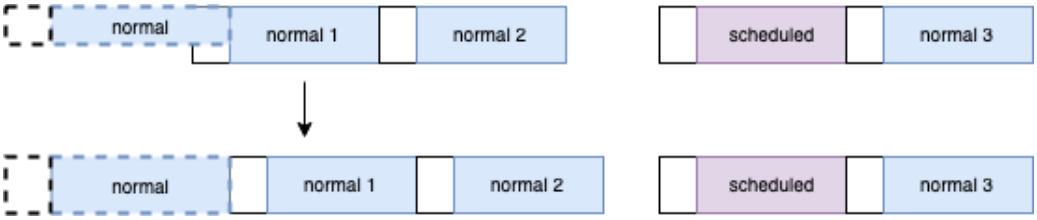

Figure 4.11: Change arrive time - longer order

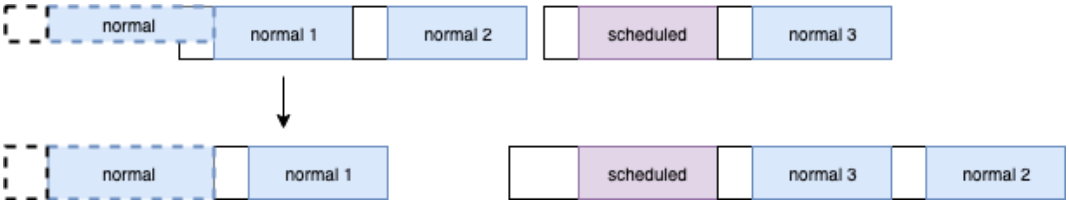

Figure 4.12: Change arrive time - longer order - collision of normal and scheduled order

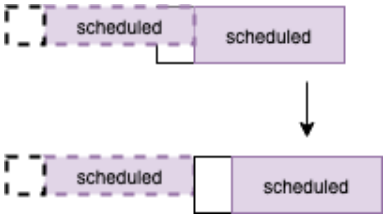

Figure 4.13: Change arrive time - longer order - collision of two scheduled orders

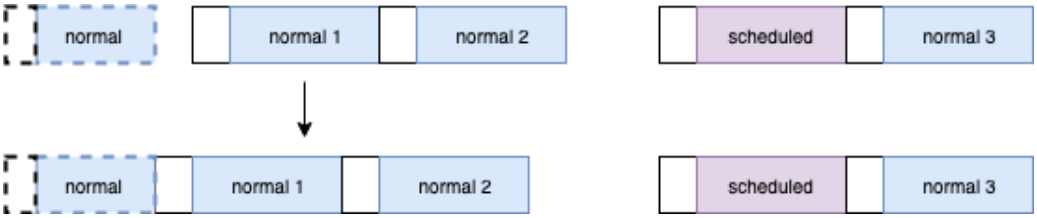

Figure 4.14: Change arrive time - shorter order

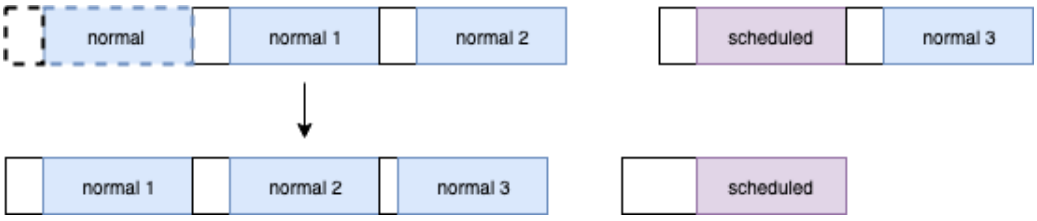

Figure 4.15: Cancel the order

5. Implementation structure

Implementation chapter reveals the overall application architecture overview. Its purpose is to provide the information using which the experienced developer can understand and further improve the project.

At first, we summarize the overall project architecture and the services involved in our solution. Then we explain the base parts of our application and how the application uses them. In the end, we focus on the implementation details of some of the application parts and explain the things that are not obvious from the project structure and conventions.

5.1 Project architecture

The whole project consists of multiple services. Back-end application, Errbit for error monitoring, Kibana for accessing the logs from all of the running services and the front-end applications. Each of the services has its own domain. They all are behind the ngnix reverse proxy which analyses incoming HTTP requests and delivers them to the desired service 5.1.

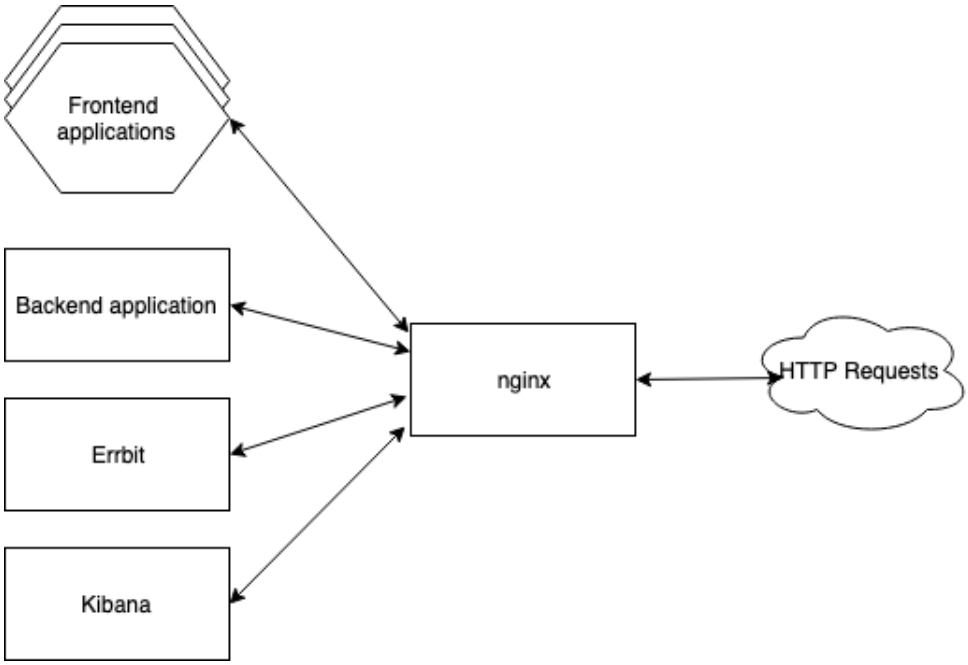

Figure 5.1: Services scheme

In the following scheme 5.2 we can see the whole application software stack described in the technical analysis. We also display the external services and APIs that our application uses. We depict the communication channels between all the parts involved.

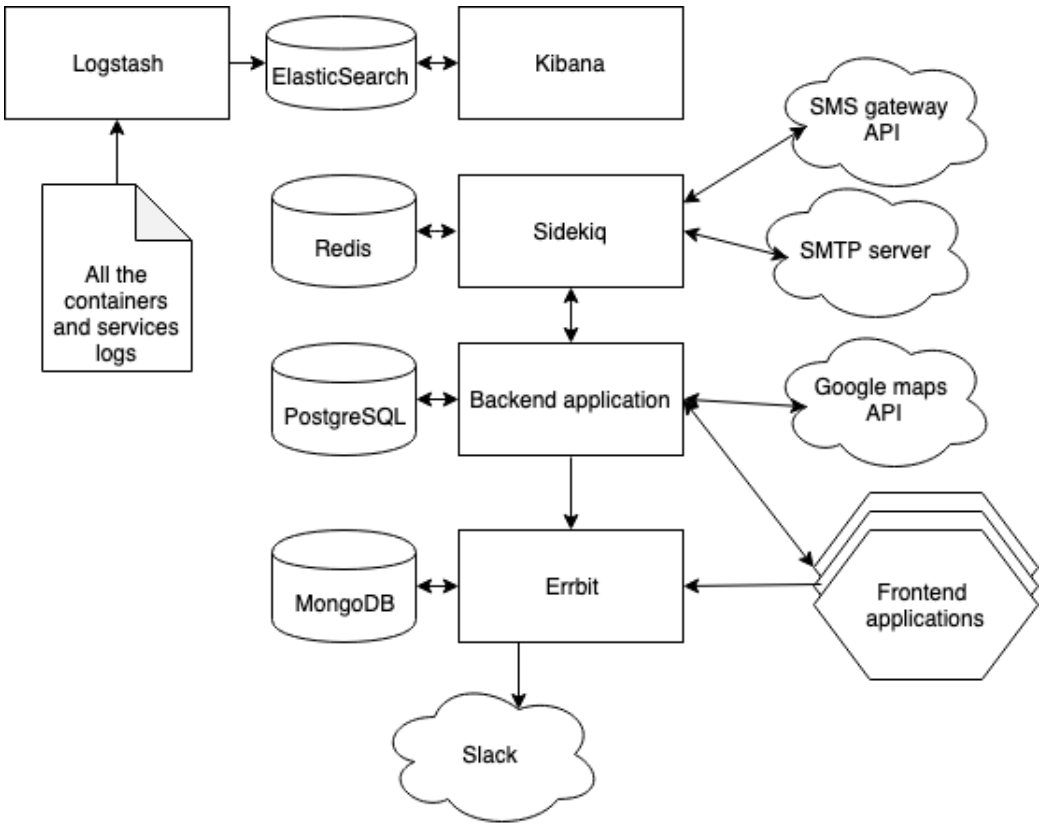

Figure 5.2: Application software stack scheme

5.2 Application architecture

In this section we describe all the key principles Ruby on Rails defines and how we work with them in our application. We describe important things the developer needs to know for further application development.

Ruby on Rails is MVC framework. We suppose the reader is familiar with the MVC[1] pattern.

Ruby on Rails framework is tightly connected with the *Convention over configuration* software design paradigm. In short it means that we as a programmer are forced to use Rails conventions otherwise the application will not run. During the application development we have to have in mind that the file and class names or the folder structure is directly connected with the whole application architecture.

5.2.1 Gemfile

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

[1] `https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller`

5.2.2 Controllers

All controllers in Ruby on Rails inherits from the *ApplicationController*. Because we are writing REST API which may define next versions in the future, we decided to add one more layer - *ApiV2Controller*. This controller base takes care about the API-specific general tasks such as authorization. All the controllers in our application are inherited from this *ApiV2Controller* controller.

Each public method (action) in the controller class usually corresponds to one API endpoint. For each resource we typically use this set of actions:

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

Each controller class is a in a separate file inside the `app/controllers` folder so you can get the overview of controllers used there.

5.2.3 Models

Ruby on Rails provide rich ORM library called *ActiveRecord*, which is the implementation of the *ActiveRecord design pattern*[2]. We use this library for the whole communication with the database.

In our application all of the models derive from the Rails *ApplicationRecord* class. Each model class corresponds one database table. In model classes we also define attribute validations and relations with other models. We can find all the application models inside the `app/models` folder.

Concerns

Concerns contain separated logic used across more models. Concern is a Ruby module which is extending the Rails *ActiveSupport::Concern* module. This module is then included in the desired models. In our application we use the concerns for authentication, user roles and notifications - all of them are used by the Customer and Employee model, thus it is convenient to have them at a separate place. Concerns are located in the `app/models/concerns` folder.

Scopes

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 from the set of employees only dispatchers.

[2] `https://en.wikipedia.org/wiki/Active_record_pattern`

5.2.4 Validators

During the order model creation we ended up with the complicated validator definition. For the better code readability we decided to separate that validator to its own class. This class lies in the `app/validators` folder as recommended by Rails.

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.

5.2.5 Database

The whole database layer is wrapped by the Ruby on Rails framework which provide us the tools to manage it.

Schema

Rails contains whole database schema with the table columns properties and table indexes in a text file `db/schema.rb`. The database schema of our application is displayed at 5.3

Migrations

All the tables in database are created and modified through the migrations. Migration is a class inherited from the Rails *ActiveRecord::Migration* base class. It specifies how is the database transformed from the original state to a desired new one. These migrations are then executed during the database setup and allow us to modify the database in the future easily.

All the migrations can be found inside the `db/migrate` folder.

Seeds

Seeds are a useful way to define the initial data. The data 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.

Seed script is located inside the `db/seeds.rb` file.

5.2.6 Workers

In general we can say that if there is a code in our application running asynchronously it is handled by workers. Each worker defines the asynchronous job.

Worker is a class which includes the *Sidekiq::Worker* module and has public method perform that is called when the job is started.

The jobs are executed in parallel so we must have the code prepared for concurrency. Each job is automatically executed again in case of error. This implies that our jobs must be designed to run repeatedly with keeping the data consistent.

In our application we use the workers for sending the SMS and emails. Besides that we use them for the favourite location indexing and the critical time order callback because the Sidekiq supports executing the jobs at specified time.

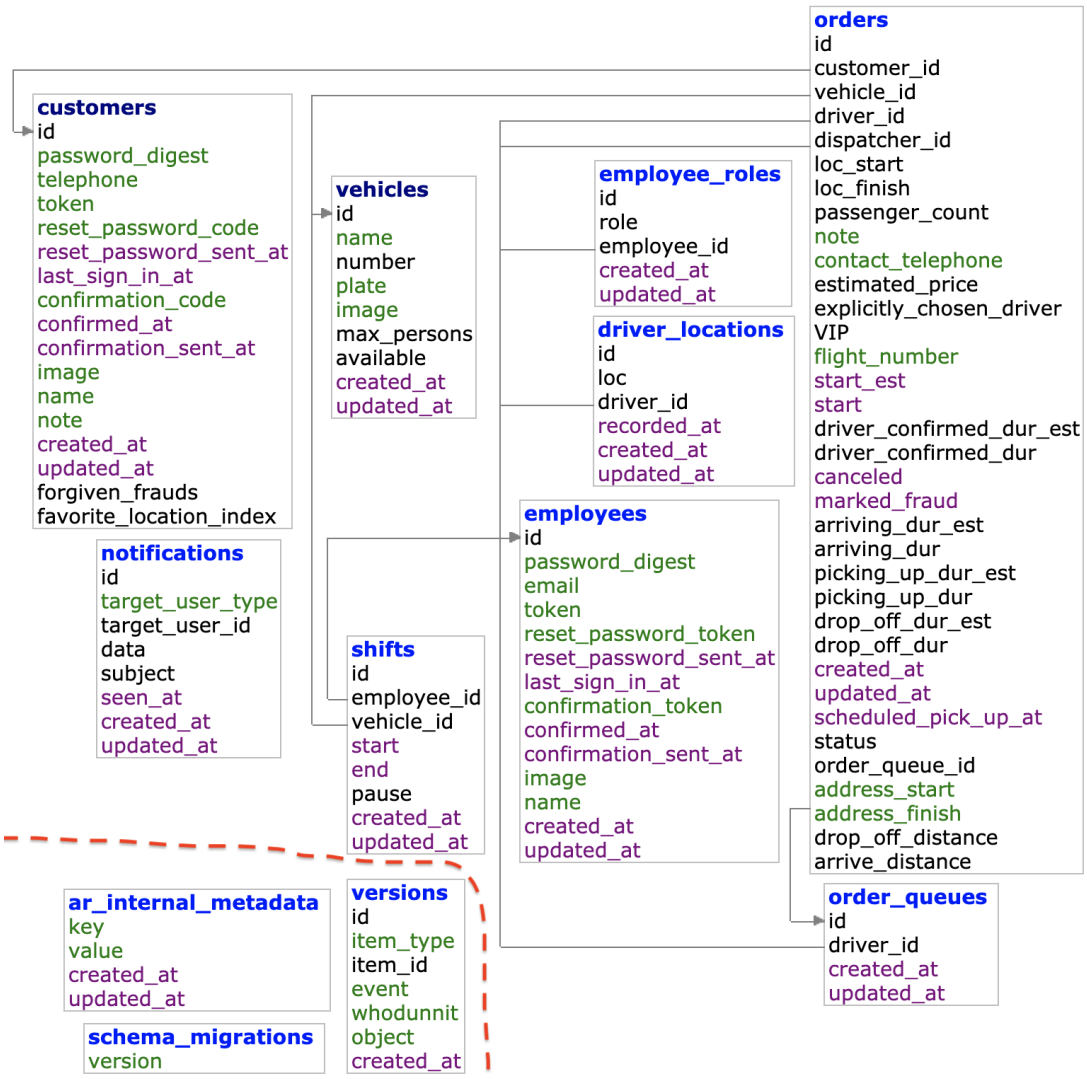

Figure 5.3: Database schema

5.2.7 Helpers

Helpers are meant for separating specific functions which are more general for the application and we want to have them separated for better code readability. Each helper is Ruby module containing the functions. We can find helpers in the `app/helpers` folder.

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

5.2.8 Mailers

Ruby on Rails has built-in support for sending emails. For each entity we derive class from the Rails *ApplicationMailer* class. Each public method (action) in this class stands for one type of email. The Rails mailer provides us with 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/mailer_entity/mail_action.*`.

In our application we use the mailers for sending the employees registration confirmations and forgotten password mails.

5.2.9 Policies

Policy classes hold the permission definitions which uses the Pundit gem. Basic principles how the policy definitions work and how they are connected with the rest of the application is described later in the 5.3.1

5.2.10 Views

In the view part we have the JBuilder templates for the REST API endpoints. Besides that the view part contains email templates.

Ruby on Rails takes care of connecting the controllers and mailers with the views. It works because we use Rails defined folder structure and file naming. For better understanding of our view architecture we are going to describe it.

Each controller/mailer has folder with the same name inside the `app/views` folder. The folder path corresponds to the class inheritance path. For each controller/mailer action that uses the view part there exists a file with a same name as the action name.

To avoid code duplication in templates we use partials. Partials are separated parts of views which can be imported in any of the view file. Partials are defined as files that start with underscore.

5.2.11 Configurations

Ruby on Rails specifies that inside the `config` folder lies all the application configurations. We describe the important ones which can be useful for future application modifications.

Initializers

Initializers handle the initialization and configuration of the modules and classes used across the application. Rails execute all the code in the `config/initializers` folder during the application start.

The important initializers in our application there are initializers for Google Maps API, Sidekiq (asynchronous jobs), Kaminari (pagination), Lograge (logs formatting), CML (our order scheduler) and CORS.

Environments

General configurations which differs for the development, testing and production environment. Specifically we can find there mail and logs configurations.

Database

Database YAML file defines which database we use on which environment. It specifies database server, credentials and the used database name.

Routes

Inside `config/routes.rb` file we define all the REST API endpoints. Ruby on Rails assumes that for each resource there exists controller with the same name. Each resource has set of default endpoints - *index*, *show*, *create*, *update* and *destroy* action.

When the application receives a request, based on this configuration file Ruby on Rails decides, which controller it handles and calls the according action method on it.

Sidekiq

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.

5.3 Specific implementation details

5.3.1 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* [3] 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[4], 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

5.3.2 Authorization

We use *Pundit* gem to help us with authorization. For each controller in `app/controllers/api/v2` there is one permissions definition class in `app/policies` folder with the according name.

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.

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

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

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

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.

5.3.3 Order scheduler

The whole order scheduler logic lies in the CML module. We can find the source codes inside the `app/lib` folder.

5.3.4 Favorite locations

5.3.5 Generators

6. API documentation

Our application communicates with the front-end applications via the REST API. As described in the technical analysis we decided to use the *Apiary* for our documentation. The documentation is available at the `https://taxiali.docs.apiary.io/`.

Besides that the documentation is also available in the standardized *API Blueprint* format [1]. located in */api_documentation.apib*.

[1] `https://apiblueprint.org/`

7. Testing

In the following chapter, we describe the 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 the third layer of testing. First is the user authentication and 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 the given feature have passed we continued testing the feature such as in the other parts of the application.

For the user part of the application we used integration tests. From our experience, the part of the application responsible for the user creation and activation is not used by the front-end developers on a regular basis thus we lose the front-end check insurance part. On top of that, the user registration and activation process is the most crucial part of the business. When there occurs error during the registration, it is likely that the user will never use our application again.

The motivation to cover the scheduler system by unit tests 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 reproducing all the edge cases took a lot of time. Besides that, we use the Google Maps API for the route calculation. This gives even for the same two points different time estimations based on current time and traffic. In the end, we decided to mock the API and write a set of tests covering all of the possible situations from the solution design ensuring the desired outputs. Based on these tests we started to implement the scheduler.

7.1 Technologies used for testing

Ruby on Rails has a variety of libraries which make the testing easy. As the testing library core we decided to use the *RSpec*[2]. This library has an 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 short, this library allows us to define multiple factories for each model in the application. When we need an instance of a model, we just use the library's factory and it will return the model instance with the data fields corresponding to the factory definition. It also allows us to specify relations between the factories so that we can easily get for example instance of the user with three orders. As random data generator for these factories, we use the gem *faker*, which helps us with generating realistically 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

In this chapter we go through the goals we set and evaluate how they have been fulfilled.

We have successfully created an application which can be used by the taxi company to handle their orders. We analysed the whole order process, specified parts involved in the process, their competencies and implemented the application which fulfils all the company's requirements.

The authentication system in our application is working as expected with all three front-end applications. In analysis we listed all the possible actions and specified the persons authorized to perform them. This was then successfully implemented and tested by the front-end. Our application doesn't send SMS messages because this service is not for free. It is fully prepared to do so, but instead of sending, it prints the tokens to the logs. When we will be ready to launch we are going to use for example the *GoSMS*[1] service, which sends the SMS just by calling their REST API endpoint.

Ordering system we created is used and tested by all front-end applications. It satisfies all front-end requirements for comfortable and smooth order processing. Besides, the system gives us detailed information about the orders. This can be later used to analyse and improve the whole ordering process.

Thanks to our order scheduler the orders are automatically assigned to currently available drivers. By using the Google Maps API for the route calculation we gained precise order duration estimations. Drivers can modify these estimations during the order process. The corrections are immediately reflected to all the affected orders. We designed the solution with emphasis on keeping the arrival time precise, especially for the orders with specified pick-up time. The whole implementation of order scheduler is covered by the unit tests which are yet another layer ensuring the orders are and will be handled and assigned correctly.

The REST API we designed was successfully used to create three front-end applications. Using the Apiary as our documentation tool was one of the best choices we have made in this project. Many times during the implementation front-end required some features that we did not have and we could not deliver at the time. In such case we just defined the interface and front-end got the mock endpoint returning the required data.

[1] `https://www.gosms.cz/`

Conclusion

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

Besides the application itself we have put together a 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 taxi companies in production and improve the services they provide. Unfortunately the recent pricing change in the Google Maps 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.

It would be useful to upgrade the authentication system so the users are able to log in on more devices simultaneously. Also implementing the automatic authentication token expiration and renewal on both front-end and back-end side would increase the overall application security.

Other possible improvements would be in the order scheduler. First of all we could separate the algorithm to service workers so we would be able to scale the number of orders processed 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

A. Installation