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

Dedication.

Title: Taxi service back-end

Author: Lukáš Březina

Department: Department of Distributed and Dependable Systems

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

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

Keywords: Taxi Ruby on Rails

Contents

JUST DRAFT! write in sentences, structure it

1. Introduction

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

Do propper introduction, mention somewhere that below:

1.1 Goals

This theses has three main goals

maybe specify more?

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

1.2 Outline

2. Requirements

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

2.1 Business requirements

JUST DRAFT!! write in sentences, structure it

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

2.2 Technical requirements

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

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

2.3 Back-end application requirements

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

2.3.1 Customers

Customers are uniquely identified by telephone number. It should support these operations:

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

Operation create

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

Directly registered customers has their own account to which they can login with password. Before they can do so, they must go through telephone number confirmation process - via sending sms with registration token. These accounts are used for orders directly from the customers via app.

Indirectly registered users exists mainly for better customer support. Most of them are created during telephone order by dispatchers.

Operation update

Operation password recovery

Operation login and logout

Operation list favorite locations

Operation list all customers and show specific customer

2.3.2 Employees

There three types of employees - admins, dispatchers and drivers.

- driver locations
- shifts

2.3.3 Vehicles

2.3.4 Orders

2.3.5 Notifications

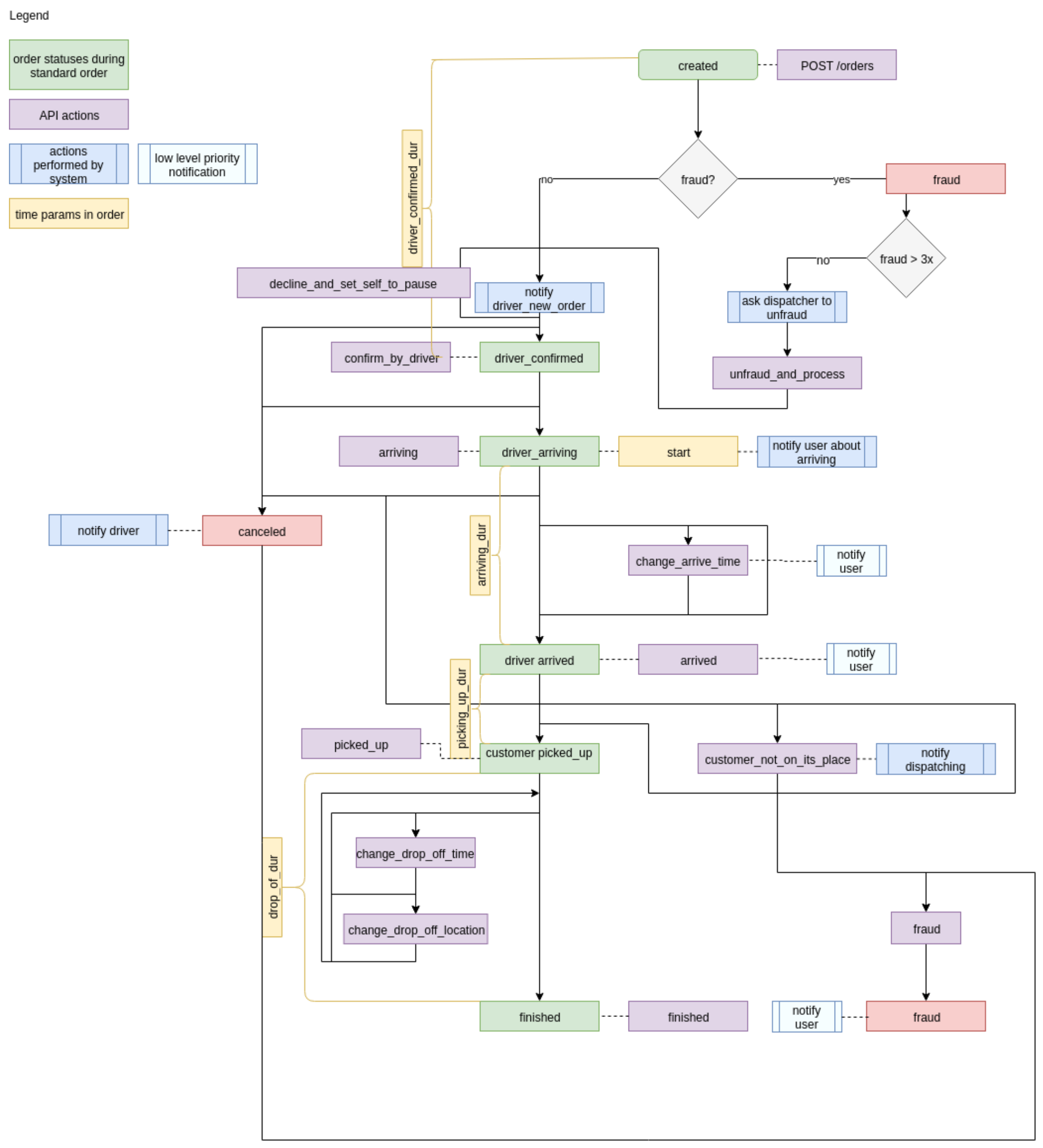

Figure 2.1: Order process scheme

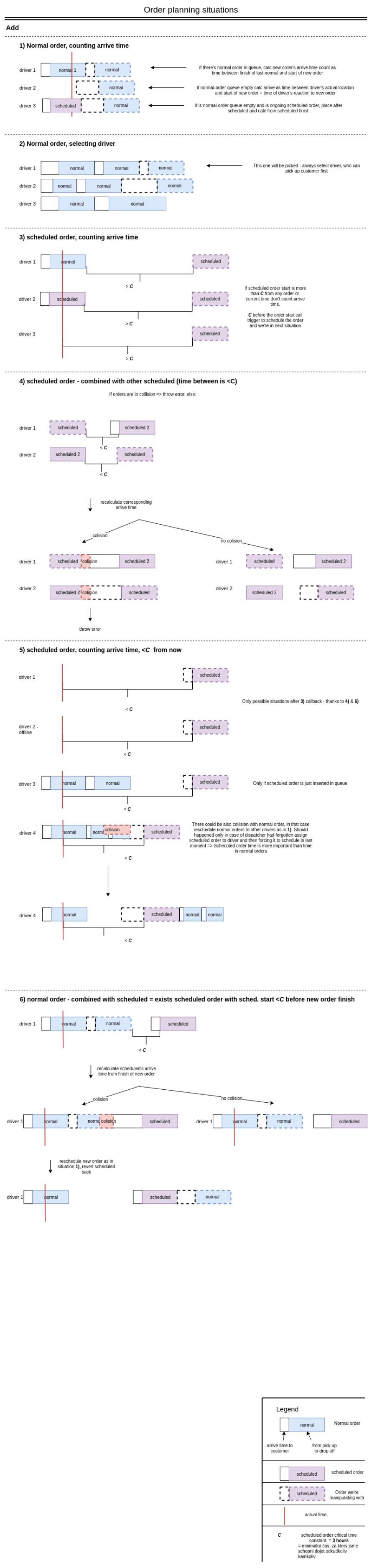
Order planning situations
Add
1) Normal order, counting arrive time
driver 1
normal 1
normal
if there's normal order in queue, calc new order's arrive time count as time between finish of last normal and start of new order
driver 2
normal
if normal-order queue empty calc arrive as time between driver's actual location and start of new order + time of driver's reaction to new order
driver 3
scheduled
normal
if is normal-order queue empty and is ongoing scheduled order, place after scheduled and calc from scheduled finish
2) Normal order, selecting driver
driver 1
normal
normal
normal
This one will be picked - always select driver, who can pick up customer first
driver 2
normal
normal
normal
driver 3
normal
normal
3) scheduled order, counting arrive time
driver 1
normal
scheduled
> C
driver 2
scheduled
scheduled
> C
driver 3
scheduled
> C
If scheduled order start is more than C from any order or current time don't count arrive time.
C before the order start call trigger to schedule the order and we're in next situation
4) scheduled order - combined with other scheduled (time between is <C)
If orders are in collision => throw error, else:
driver 1
scheduled
scheduled 2
< C
driver 2
scheduled 2
scheduled
< C
recalculate corresponding arrive time
colision
no colision
driver 1
scheduled
colision
scheduled 2
driver 2
scheduled 2
colision
scheduled
throw error
driver 1
scheduled
scheduled 2
driver 2
scheduled 2
scheduled
5) scheduled order, counting arrive time, <C from now
driver 1
scheduled
< C
Only possible situations after 3) callback - thanks to 4) & 6)
driver 2 - offline
scheduled
< C
driver 3
normal
normal
scheduled
< C
Only if scheduled order is just inserted in queue
driver 4
normal
norm
colision
scheduled
< C
There could be also collision with normal order, in that case reschedule normal orders to other drivers as in 1). Should happened only in case of dispatcher had forgotten assign scheduled order to driver and then forcing it to schedule in last moment => Scheduled order time is more important than time in normal orders
driver 4
normal
scheduled
normal
normal
< C
6) normal order - combined with scheduled = exists scheduled order with sched. start <C before new order finish
driver 1
normal
normal
scheduled
< C
recalculate scheduled's arrive time from finish of new order
colision
no colision
driver 1
normal
norma
colision
scheduled
driver 1
normal
normal
scheduled
reschedule new order as in situation 1), revert scheduled back
driver 1
normal
scheduled
normal
Legend
normal
Normal order
arrive time to customer
from pick up to drop off
scheduled
scheduled order
scheduled
Order we're manipulating with
actual time
C
scheduled order critical time constant. = 3 hours = minimální čas, za který jsme schopni dojet odkudkoliv kamkoliv

Figure 2.2: Order planning situations

3. Technical Analysis

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

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

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

Advantages:

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

Disadvantages:

- it is more difficult to set it up than PHP

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

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

Add citation

3.1.2 PosrtgreSQL

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

3.1.3 Sidekiq

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

3.2 Tools stack

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

Include links to all libraries

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

3.2.1 Apiary

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

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

3.2.2 Docker and Docker-compose

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

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

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

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

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 research we have found out, that for that purpose was made set of Docker images - jwilder's ngnix-proxy, which does exactly what we needed - including automated management of HTTPS certificates.

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

3.4 Errbit

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

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

[7]from our observations during development

3.5 Monitoring

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

The logging stack that we use consists of

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

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

4. Analysis

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

4.1 Orders

5. Implementation

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

6. Testing

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

7. Evaluation

Conclusion

Bibliography

J. Anděl. *Základy matematické statistiky*. Druhé opravené vydání. Matfyzpress, Praha, 2007. ISBN 80-7378-001-1.

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

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

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

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment