Company Ride

PROJECT BOOK

PROJECT BY:

Vladislava Leykin, *317810414*

Ruslan Ibragimov, *320956758*

ADVISOR:

Dr. Osnat Mokryn

PROJECT NUMBER: *150203* YEAR: *2015*

Table of Contents

Introduction

CompanyRide is an application that helps people to share their everyday rides to work or home or anywhere else. In this document we will describe all aspects of the project, its logic and architecture.

Motivation

Traffic congestion are a major daily headache of the modern life. It badly affects our health and have negative impact on the environment. Also increased travel times impose costs on the economy. Traffic congestions are growing worse in all large and growing metropolitan areas across the world. City centers and industrial areas suffers the most at the morning and evening hours while almost everybody drive the car alone.

New road development takes years to complete, revolutionary public transportation solutions are not here today, so reducing number of vehicles may be the main solution for traffic congestion.

Car sharing solutions are not widely used so far. So we decided to explore this topic and create the solution that will likely be adopted and used as a main car sharing solution in the future.

Market research

At the first stage of the project we explored the topic of car sharing in Israel. We found some car sharing solutions and tried to understand why people don't use them.

Some disadvantages that we noticed in current solutions:

✗ **List of rides instead of smart matching** – if someone needs to search for a ride by himself there can be too many options. People get tired and disappointed while system can check matches in way that is more efficient.

✗ **Open to large public** – many people sympathize the idea of reducing traffic and contributing to a better environment, but they not ready to advertise their contact information, or propose a ride to just **everyone**.

✗ **No further support after matching** – After matching is done some applications provide telephone number to discuss pick up details, redirect to Messenger application or other kind of message exchange. Anyway there is no simple interface to discuss all the details. If there is more than one hitcher it burdens the driver.

Project targets

We decided to build an application that is free of the above disadvantages:

✓ **Smart matching algorithm -** Matching algorithm releases people from exhausting searches. It's build to create long term driver-hitcher relationships, so people would be exempted of searching companions every day.

✓ **Free of contact information sharing** – All ride relevant communication is done within application: After matching occurs driver and hitcher get introduced by viewing each other user profile. Afterwards they agree to share a ride together, driver sets a place and a time of pick up. When the ride is active, people can send each other short messages to inform the status.

✓ **Ride management instruments** – One of our main differentiators in providing support for a ride after a matching has occurred. Rides are presented as calendar events. All ride relevant information is presented in one place. Map view of all ride points: pick-ups, drops.

As all the targets was specified CompanyRide system was specified and build.

Application usage

In this paragraph, we describe a system in action by the example of Sam, who lives in Rishon-le-Zion and rides every morning to a job at Ramat-ha-Hayal high-tech area.

After Sam passed the registration process and verified his email address the system created a profile that identify Sam throughout the system. It contains personal information, Sam's experience in our application (e.g. rating) etc.

Sam is now an active user and can start share a ride and enjoy all features of the application. Sam enters his "Hitch Request" and "Ride Proposal" events to the system according to his habits and needs.

Weekday	Habit / need	Events to be created
Sunday	Sam rides to work around 8 AM	Weekly "Ride proposal" event around 8 AM to Ramat-Ha-Hayal
Monday	Sam goes to a gym in the morning and rides to work around 10 AM	Weekly "Ride proposal" event around 10 AM to Ramat-Ha-Hayal
Tuesday	Sam's wife usually takes a car to make shopping after work. Sam need a hitch	Weekly "Hitch Request" event around 8 AM
Wednesday	Sam goes to work in the Hertzlia office around 8 AM	Weekly "Ride proposal" event around 8 AM to Hertzlia
Thursday	Sam has a meeting in Jerusalem in the middle of the day	One-time "Hitch Request" event around 12 AM

Our system looks at all the "Hitch Request" and "Ride proposal" events and finds matches between events. For detailed explanation, see Matching process.

After a while Sam gets a message that his Wednesday "Ride proposal" got a match to a hitcher Lucy Miller. Sam viewing Lucy's user profile, and decides to share a ride with her. Sam enters time and location of Lucy's pick up.

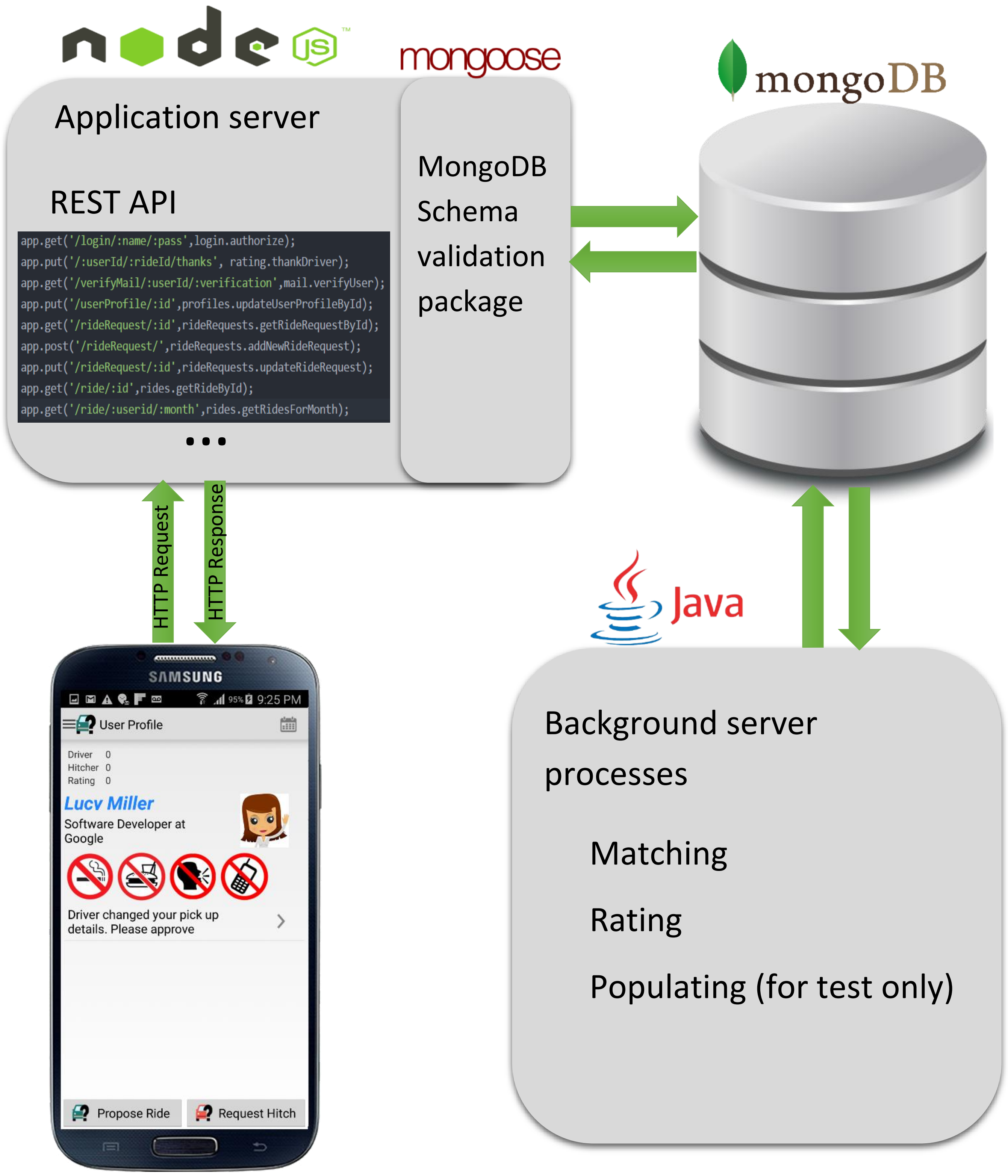
Lucy gets the application message about the matching. If Lucy feels convenient riding with Sam and the pick-up time and location are fine, she approves a ride.

Now, when ride is active, Sam and Lucy can send messages to each other related to the upcoming ride. When Lucy gets to the location of a pick-up she notifies Sam by sending "I'm at the point".

After a ride Lucy receives a message to thank Sam for a ride. When Lucy sends her "Thank you", 10 points are added to Sam's rating.

From now on Sam and Lucy share a ride on each Wednesday morning to Hertzlia.

High level design



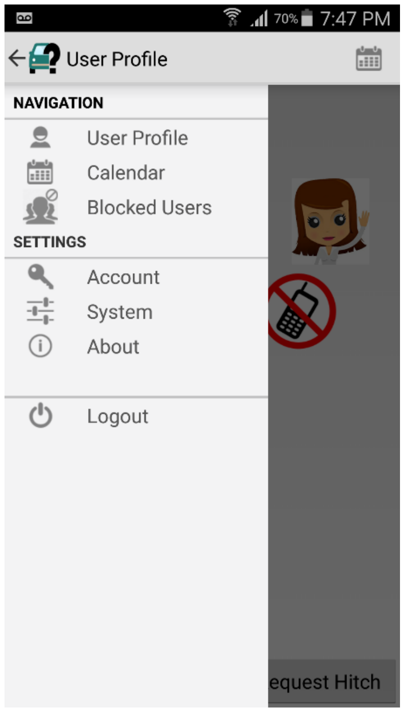
```
app.get('/login/:name/:pass',login.authorize);
app.put('/:userId/:rideId/thanks', rating.thankDriver);
app.get('/verifyMail/:userId/:verification',mail.verifyUser);
app.put('/userProfile/:id',profiles.updateUserProfileById);
app.get('/rideRequest/:id',rideRequests.getRideRequestById);
app.post('/rideRequest/',rideRequests.addNewRideRequest);
app.put('/rideRequest/:id',rideRequests.updateRideRequest);
app.get('/ride/:id',rides.getRideById);
app.get('/ride/:userid/:month',rides.getRidesForMonth);
```

Client side

Client side in our project is a native Android application. The next paragraph describes the main practices and method we used in application implementation, and the following paragraphs describe the logic of CompanyRide Application along with application screenshots.

Implementation

- **Fragments rather than activities** – Most of the screens in CompanyRide application were implemented as fragments rather than activities. Fragments are modular section of an activity, which has its UI, lifecycle, event handlers and logic. Activity hosts fragments showing /hiding them according to the predefined logic. In CompanyRide project we have defined several groups of fragments that are logically tied together and share common parent activity. Thus the MainScreen activity hosts userProfileFragment, calendarFragment and many other fragments. Such model allowed us to use Navigation Drawer as a main menu of an application.
  Fragments has another advantage – they can be embedded in more than one activity, contributing to code reuse and modularity. All the dialogs are implemented as fragments in our application and are used by several activities.



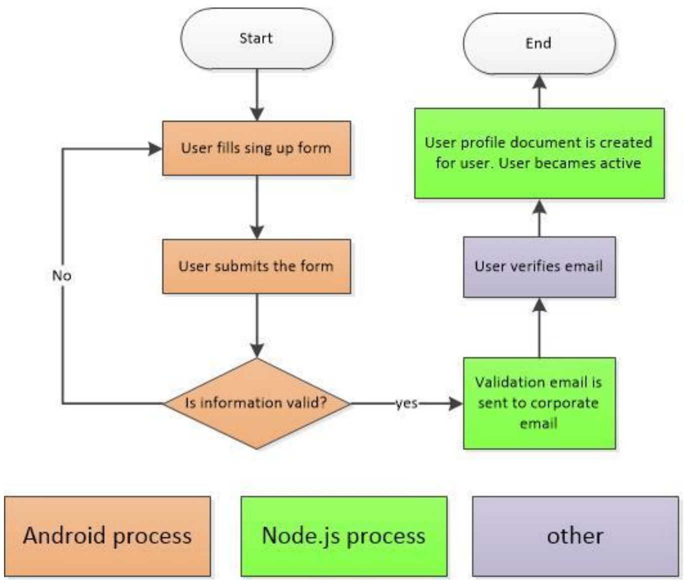
- **AsyncTask for server communication –** Each application that has UI needs to be full time responsive. Thus, time consuming operation, like server communication, can't be done in the main thread. Each HTTP request was performed using the Android AsyncTask, the task that runs in background in a separate thread that has no influence on a GUI thread.

- **ListView and ListAdapter –** When multiple objects have to be shown (like multiple hitchers or messages) ListView UI object need to be used. ListAdaptor is a class to be extended to store and display the list objects. The ListView is automatically scrollable, and ListAdapter has built-in mechanism of recycling instances of no longer visible list items, which makes using ListViews very efficient.

- **Reusing layouts –** Android UI is stored in XML form and is dynamically built at the runtime. Android offers different ways of including or inflating UI objects into layout, which allows to use once defined UI module again and again.

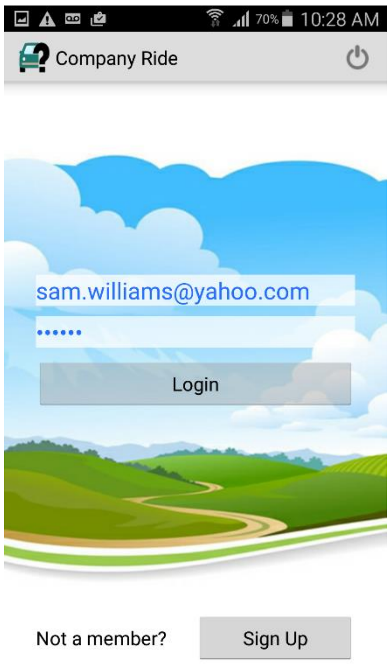
Sign up and Log in

User needs to pass sign up procedure to become a member of CompanyRide. The following flow diagram describes the procedure.

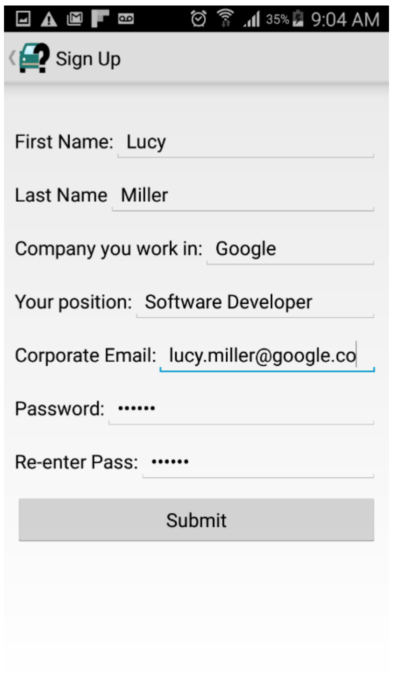


Flow diagram 1 - Sign up procedure

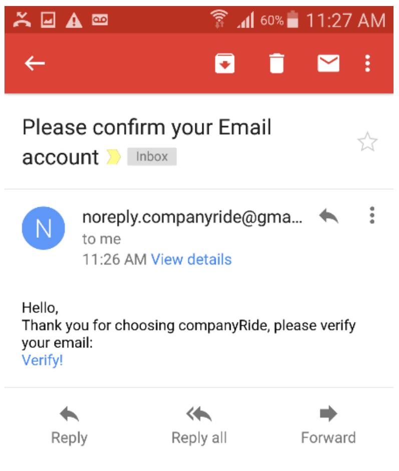
Below you can see log in, sign up screenshots and a screenshot of email sent by CompanyRide



sam.williams@yahoo.com
••••••



First Name: Lucy
Last Name Miller
Company you work in: Google
Your position: Software Developer
Corporate Email: lucy.miller@google.co
Password: ••••••
Re-enter Pass: ••••••



Please confirm your Email account

noreply.companyride@gma...
to me
11:26 AM View details

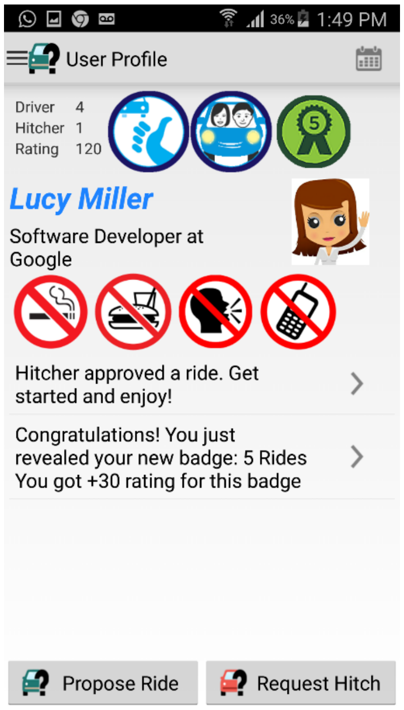
Hello,
Thank you for choosing companyRide, please verify your email:
Verify!

User and User Profile

Information about user is saved in two different form (documents – see MongoDB technology).

User document holds all the private details of a user (e.g. password, date of validation). It's used to identify the user for server. Most of this information passed to server at the sign up stage. For detailed information saved in user document see Figure 2 - User document schema.

User profile holds user activity information as a CompanyRide member (e.g. number of rides as hitcher, number of rides as driver, special requests for hitchers). People can be introduced by looking at each other user profiles. At the screenshot to the right you can see user profile view of Lucy Miller. For detailed fields of user profile document see Figure 3 - User profile document schema.



Driver 4
Hitcher 1
Rating 120

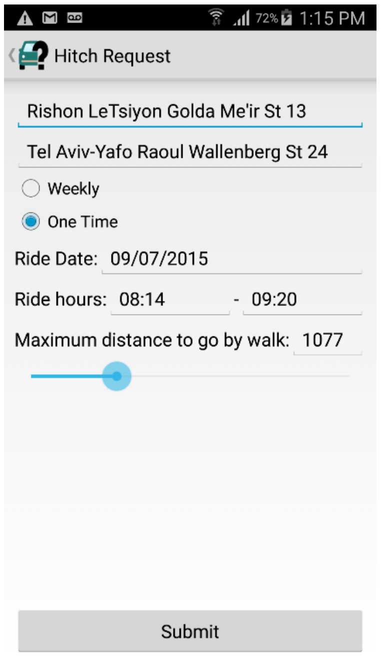
Lucy Miller
Software Developer at Google

Hitcher approved a ride. Get started and enjoy!

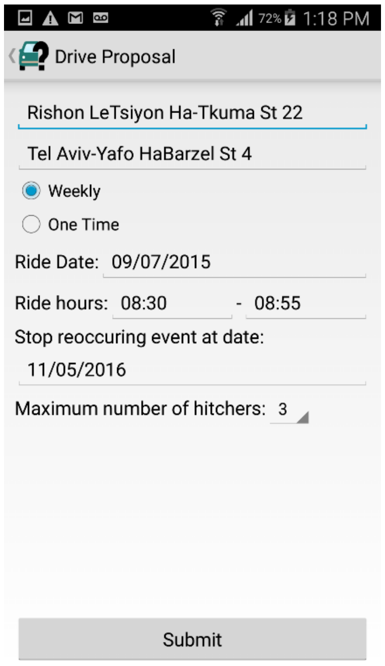
Congratulations! You just revealed your new badge: 5 Rides You got +30 rating for this badge

Ride request creation

Both "Hitch Request" and "Ride proposal" events in our system are called Ride requests. The lack of need to distinguish between "Hitch Request" and "Ride proposal" seemed to be the right thing:



Rishon LeTsiyon Golda Me'ir St 13
Tel Aviv-Yafo Raoul Wallenberg St 24
Weekly [✓ One Time]
Ride Date: 09/07/2015
Ride hours: 08:14 - 09:20
Maximum distance to go by walk: 1077



Rishon LeTsiyon Ha-Tkuma St 22
Tel Aviv-Yafo HaBarzel St 4
[✓ Weekly] One Time
Ride Date: 09/07/2015
Ride hours: 08:30 - 08:55
Stop reoccuring event at date: 11/05/2016
Maximum number of hitchers: 3

Two events are more similar than different – both are just calendar events, the only difference is that "Hitch Request" has "Maximum distance to go by walk" field and "Ride proposal" event has "Maximum number of hitchers" field. The schemaless flexibility of MongoDB has allowed us to save both events in the same collections and prevent many code replication (e.g. in update procedure).

Ride matching and two side approval

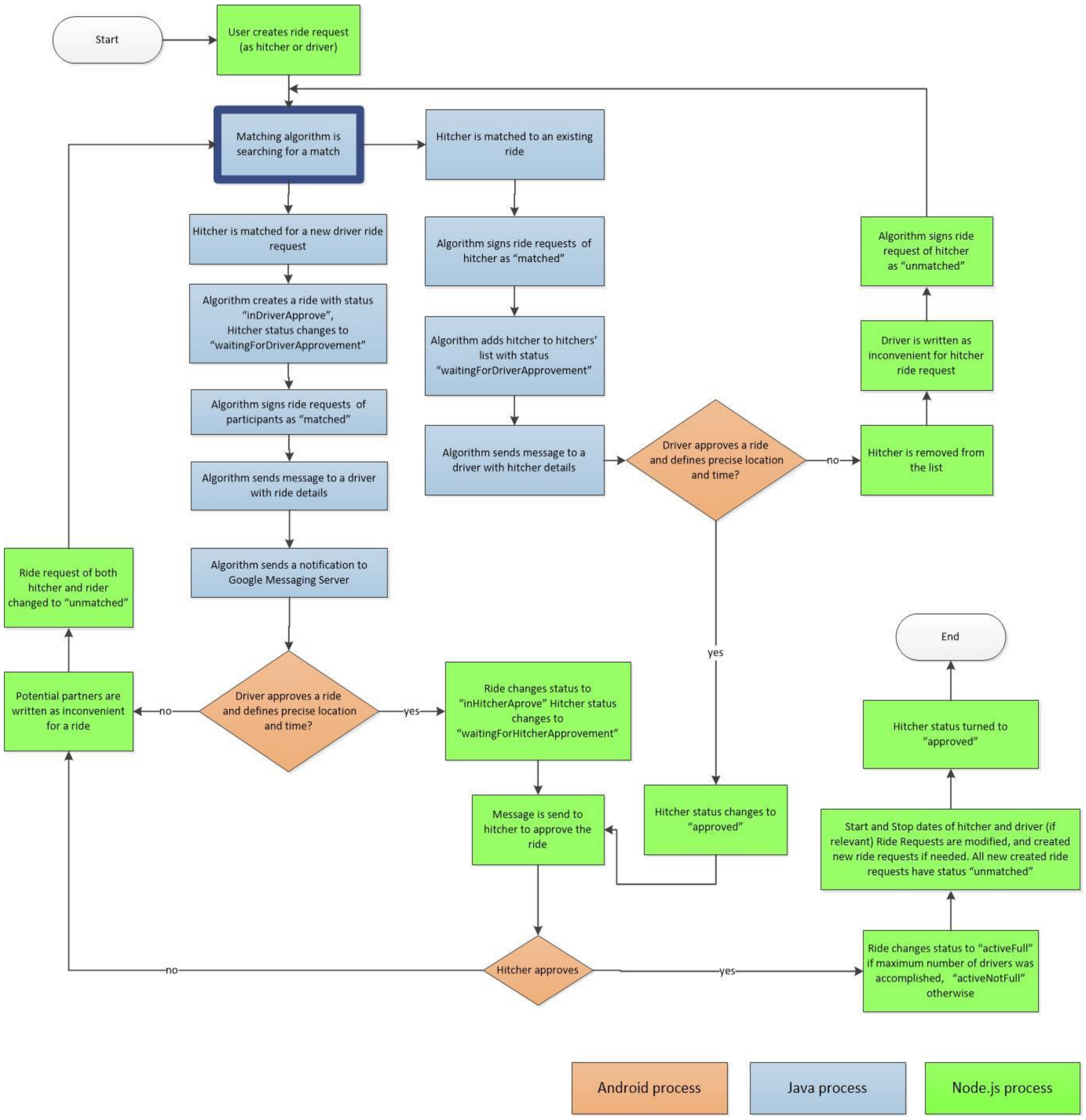
After people had entered their Ride Requests to the system, matching process at the server begins to find matches for the events. See below flow diagram of ride matching and agreement.

After the matching was found driver and hitcher need to agree to be ride companions. The agreement process is done via application, in CompanyRide people don’t share their private contact information. The steps of agreement:

- The driver is first to know, he/she got notified by the server (with the help of Google Cloud Messaging) about the upcoming ride. Driver is introduced to hitcher user profile.
  In case of disapproval, hitcher will never know the ride was ever matched and hitcher will no longer appear as a potential hitcher for this specific ride.
  In case of approval driver will set pick up time and location and drop location for the hitcher.
- At the next step hitcher is notified about the upcoming ride. He/she needs to approve the driver and all the pick-up and drop details. If he/she disapproves, new ride is removed as never happened, and the driver will no longer be proposed for this specific ride request. In case of approval ride becomes active.

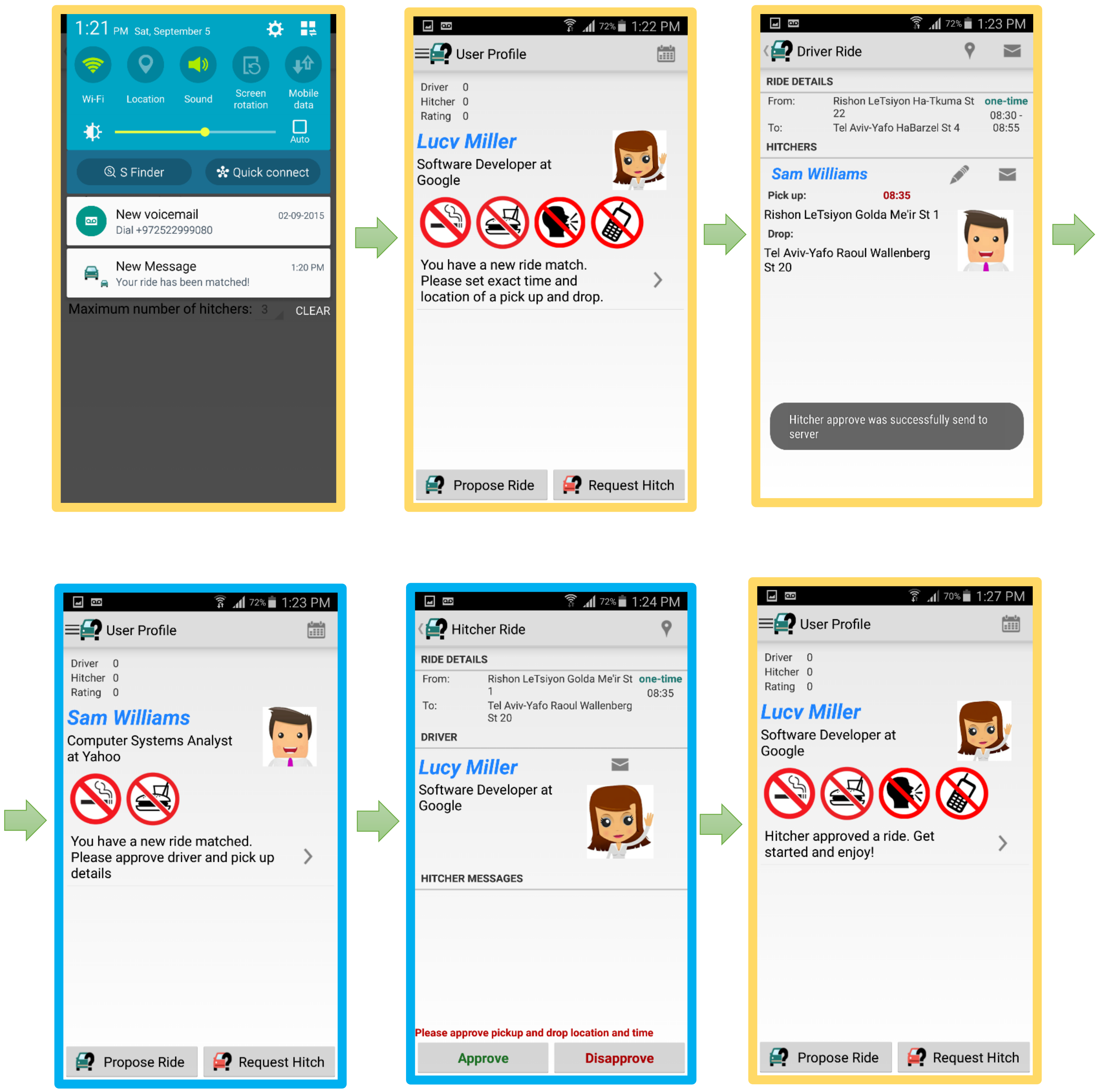
The following flow diagram describes the logic of ride matching and agreement process in details. The blue processes implemented in Java, and green processes implemented in Node.js.

Matching algorithm, surrounded by deep blue frame is described in Matching process paragraph.


Start
User creates ride request (as hitcher or driver)
Matching algorithm is searching for a match
Hitcher is matched to an existing ride
Hitcher is matched for a new driver ride request
Algorithm signs ride requests of hitcher as "matched"
Algorithm creates a ride with status "inDriverApprove", Hitcher status changes to "waitingForDriverApprovement"
Algorithm adds hitcher to hitchers' list with status "waitingForDriverApprovement"
Algorithm signs ride requests of participants as "matched"
Algorithm sends message to a driver with hitcher details
Algorithm sends message to a driver with ride details
Algorithm sends a notification to Google Messaging Server
Driver approves a ride and defines precise location and time?
no
Hitcher is removed from the list
Driver is written as inconvenient for hitcher ride request
Algorithm signs ride request of hitcher as "unmatched"
yes
Ride request of both hitcher and rider changed to "unmatched"
Potential partners are written as inconvenient for a ride
no
Driver approves a ride and defines precise location and time?
yes
Ride changes status to "inHitcherAprove" Hitcher status changes to "waitingForHitcherApprovement"
Message is send to hitcher to approve the ride
Hitcher status changes to "approved"
End
Hitcher status turned to "approved"
Start and Stop dates of hitcher and driver (if relevant) Ride Requests are modified, and created new ride requests if needed. All new created ride requests have status "unmatched"
no
Hitcher approves
yes
Ride changes status to "activeFull" if maximum number of drivers was accomplished, "activeNotFull" otherwise
Android process
Java process
Node.js process

Flow diagram 2 - Matching and agreement process procedure

The below figure shows the matching and agreement procedure in screenshots. Driver screenshots are with yellow frame and the blue ones are hitcher's.


1:21 PM Sat, September 5

Wi-Fi Location Sound Screen rotation Mobile data

Auto

S Finder Quick connect

New voicemail 02-09-2015
Dial +972522999080

New Message 1:20 PM
Your ride has been matched!

Maximum number of hitchers: 3 CLEAR

72% 1:22 PM

User Profile

Driver 0
Hitcher 0
Rating 0

Lucy Miller
Software Developer at Google

You have a new ride match. Please set exact time and location of a pick up and drop.

Propose Ride Request Hitch

72% 1:23 PM

Driver Ride

RIDE DETAILS

From: Rishon LeTsiyon Ha-Tkuma St 22 one-time 08:30 - 08:55
To: Tel Aviv-Yafo HaBarzel St 4

HITCHERS

Sam Williams
Pick up: 08:35
Rishon LeTsiyon Golda Me'ir St 1
Drop:
Tel Aviv-Yafo Raoul Wallenberg St 20

Hitcher approve was successfully send to server

72% 1:23 PM

User Profile

Driver 0
Hitcher 0
Rating 0

Sam Williams
Computer Systems Analyst at Yahoo

You have a new ride matched. Please approve driver and pick up details

Propose Ride Request Hitch

72% 1:24 PM

Hitcher Ride

RIDE DETAILS

From: Rishon LeTsiyon Golda Me'ir St 1 one-time 08:35
To: Tel Aviv-Yafo Raoul Wallenberg St 20

DRIVER

Lucy Miller
Software Developer at Google

HITCHER MESSAGES

Please approve pickup and drop location and time

Approve Disapprove

70% 1:27 PM

User Profile

Driver 0
Hitcher 0
Rating 0

Lucy Miller
Software Developer at Google

Hitcher approved a ride. Get started and enjoy!

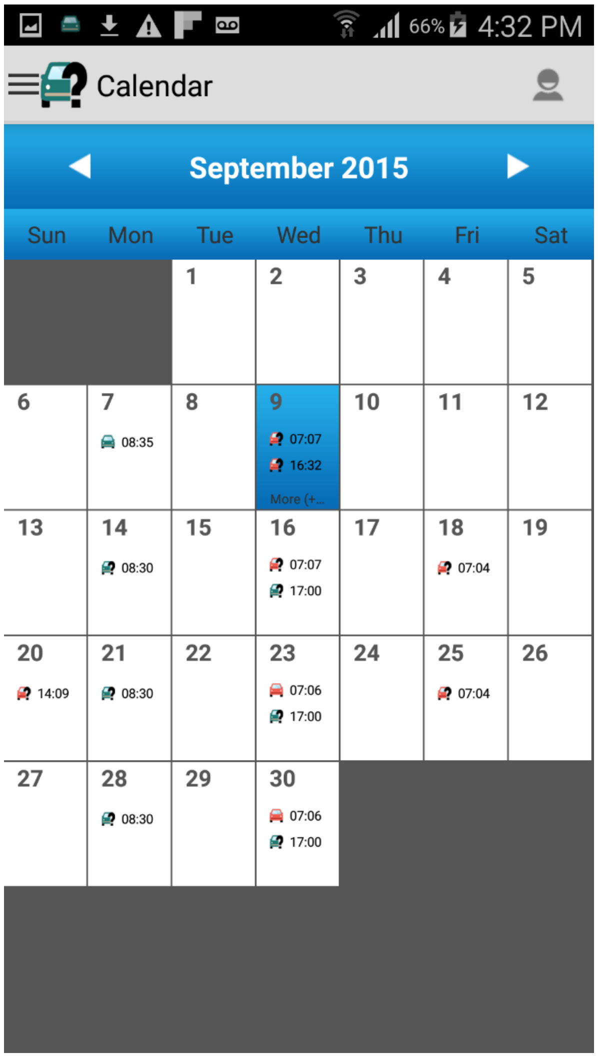
Propose Ride Request Hitch

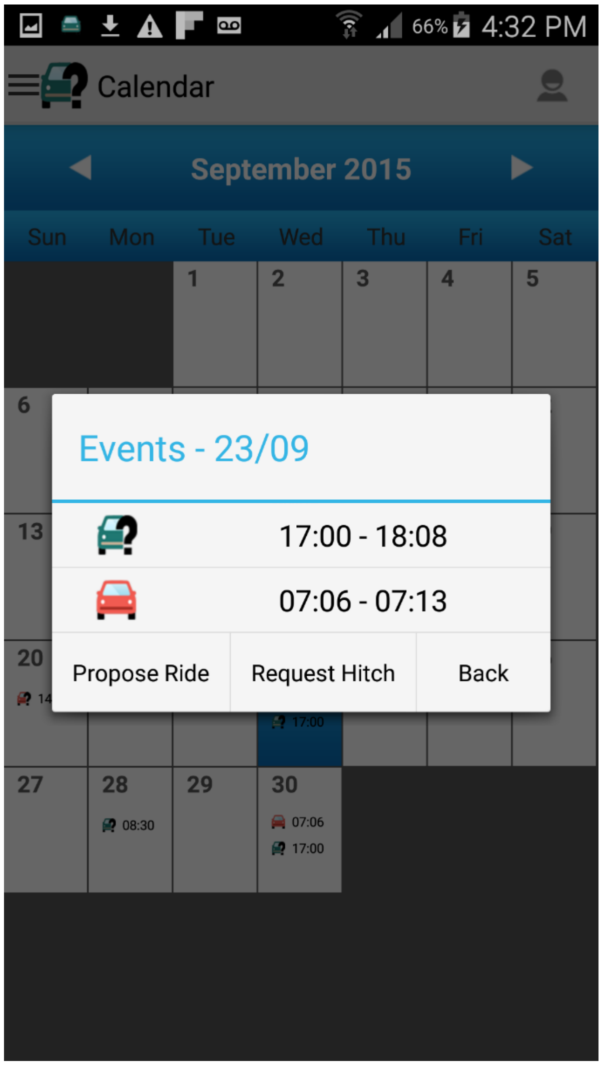
Figure 1 - Two-side driver hitcher approval after a matching

Ride management tools

Calendar

“Hitch request”-s and “Ride proposal”-s, also called ride requests in our system, are first of all events, one-time or weekly, having start and stop dates. So are the rides. One of the most usable features of the application is calendar showing all rides and ride requests. The screenshots below shows the calendar view and the dialog that is shown after clicking at some date.





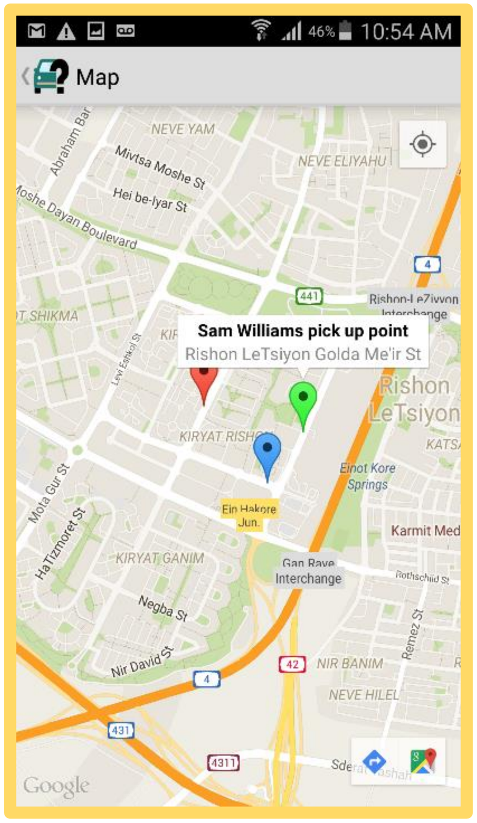
Map

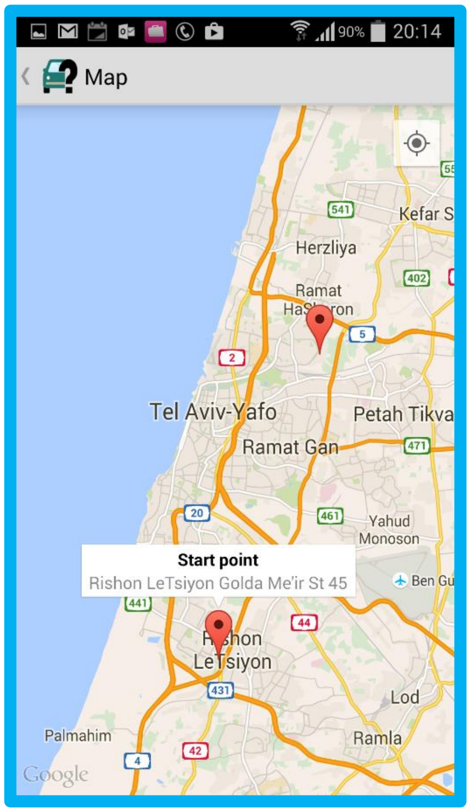
Another very useful feature is a map.

Driver view of a map shows start, end point of a ride and pick up and drop locations of all hitchers. Map helps driver be focused on driving and not on searching a way to get to his hitchers. From map view driver can switch to google directions quickly for navigation

Hitcher view of a map shows only hitcher pick up and drop locations. Also the “Hitcher Ride” screen, shown at Figure 1 - Two-side driver hitcher approval after a matching, shows hitcher pick up location in “From” field, and a drop location in “To” field. Hitcher is never told the locations of driver or other hitchers. So the privacy of all ride members is observed.

Driver screenshots are with yellow frame and the blue ones are hitcher’s.

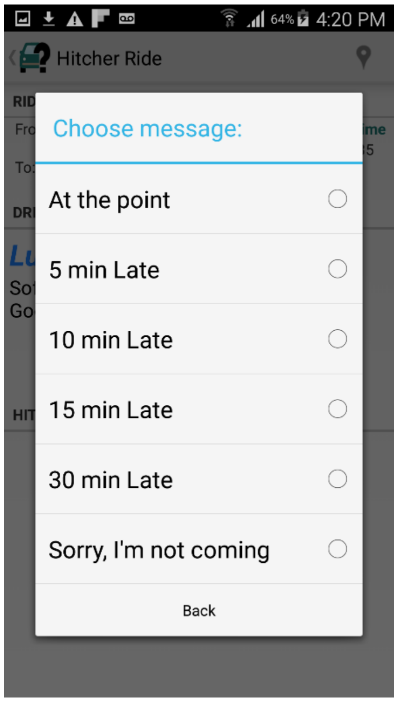




Messages

CompanyRide don't ask people to share their personal contact information, instead our system provides facilities to conduct satisfying communication. One of them was the two-side approval process described in Flow diagram 2 - Matching and agreement process procedure.

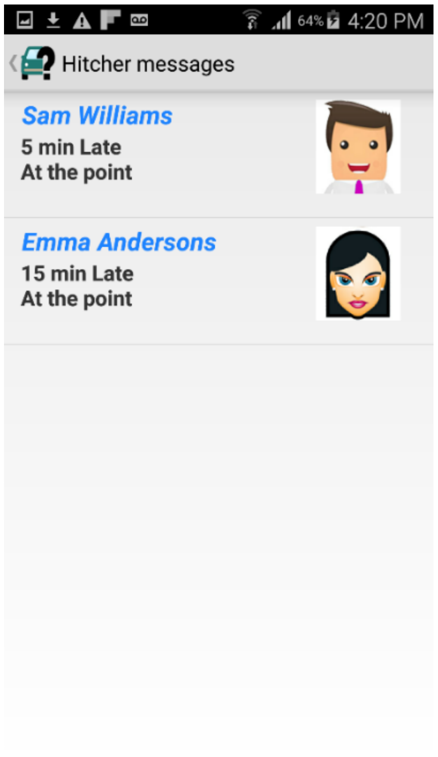
When the ride approaches ride members can exchange build-in messages, shown to the right. Using built in messages instead of typing is very important for safe driving.



Messages that are sent by hitchers are seen by driver only, and messages from driver are sent to each hitcher separately. Driver view all his messages from hitchers in one place – in a special "Hitcher messages" screen. Hitcher see messages from driver in "Hitcher Ride" screen along with all ride information. See below screenshots.

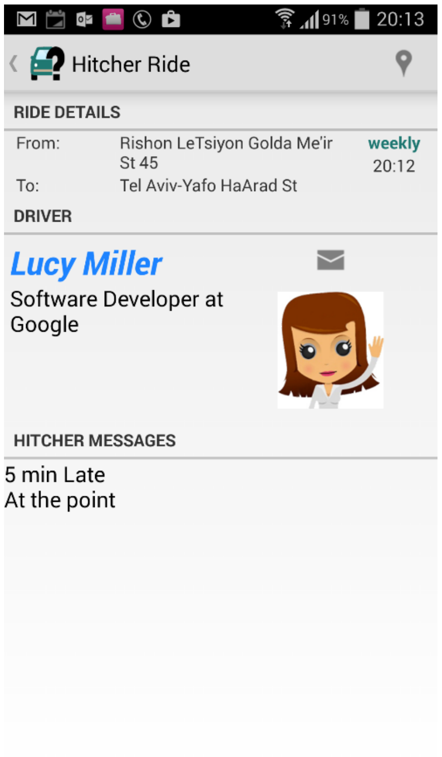
When new message arrives ride member is notified by GCM server about a new message. More about it in the next paragraph.

Messages are cleared at the day of the ride after the ride has occurred. Message clearing is done as a part of the rating procedure, described in Rating process paragraph.


Hitcher messages

Sam Williams
5 min Late
At the point

Emma Andersons
15 min Late
At the point


Hitcher Ride

RIDE DETAILS

From: Rishon LeTsiyon Golda Me'ir St 45 weekly 20:12
To: Tel Aviv-Yafo HaArad St

DRIVER

Lucy Miller
Software Developer at Google

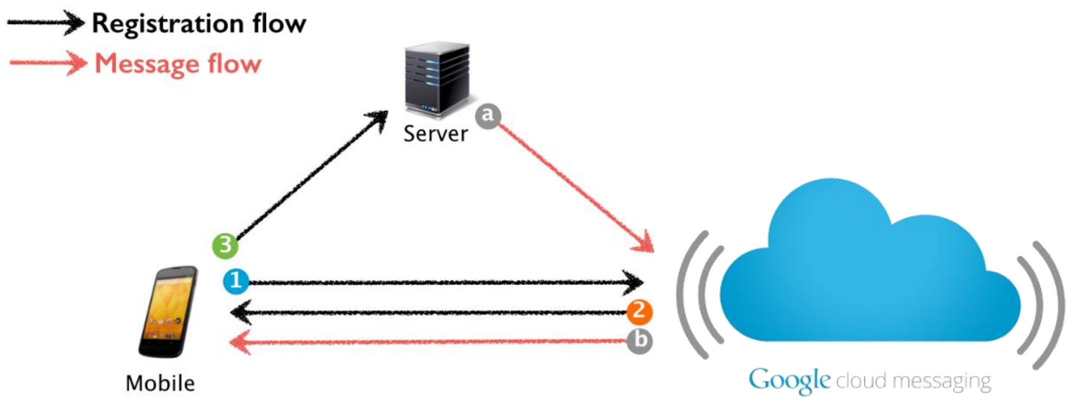
HITCHER MESSAGES

5 min Late
At the point

Notifications

To notify a user about a matched ride or about a new message that has come we use Google Cloud Messaging technology. GCM is a service that allows to send data from server to Android application. The service handles message queueing and storing in case the target device is offline.

Message delivery is possible to a single device, to device group or to the topic. In CompanyRide project we have chosen to work with topics, as there may be users logged in with more than one device on one hand, and multiple users using the same device on the other hand. We use user profile id as a topic id. It is sufficient that user was logged in to the device and didn't performed logout lately (logout makes unsubscribe from topic). The android application don't need to run in order to receive messages. The system will wake up the application via a mechanism called Intent Broadcast when the message arrives.



There are three entities in Google Cloud Messaging communication: Google messaging server, CompanyRide server and Android application running CompanyRide application.
The registration flow is as follows:

- At the time user logs in, the device receives the communication token from Google messaging server.
- The device subscribes itself with the received token to a topic named by user profile id that has just logged in.
- In case the server pushes notification to single device (by its token), device need to deliver its token to the CompanyRide server. As long as we use topic communication, this step is not needed.

The message flow is as follows:

- The Android application is running a service listening to the upcoming events.
- The server sends the message to Google messaging server topic.
- Google messaging server delivers the message to all devices registered to the topic.

Server side

Server side consists of:

- Node.js application server that provides REST API for clients
- Mongoose package integrated to Node.js that validates schema of new information inserted to MongoDB
- MongoDB NoSQL document database as a main storage
- Background processes that perform matching of rides and rating of users that participated in a ride.

Node.js Application Server

Node.js application server exposes server API for clients (Android clients in our project). It is responsible to carry all clients' requests and change the server state according to system logic. Node.js server is also responsible to validate the incoming data to prevent inconsistent, wrong or corrupted information being inserted to database.

Node.js technology

Node.js is an open source environment for building fast network application on server-side. The principle of Node.js in no waiting for a time-consuming blocking I/O operation, but in responding to asynchronous events. Such operation model makes Node.js very efficient and perfect for data-intensive real-time applications that run across distributed devices.

Node.js is running in a single-thread, which means it serves one event at the single point of time, thus all node.js operations should be short and time-efficient. Upcoming events (new http requests or asynchronous callback events) are managed in a queue form. Node.js non-blocking, asynchronous model allows it to be efficient within single thread and keeps it away of multi-thread troubles.

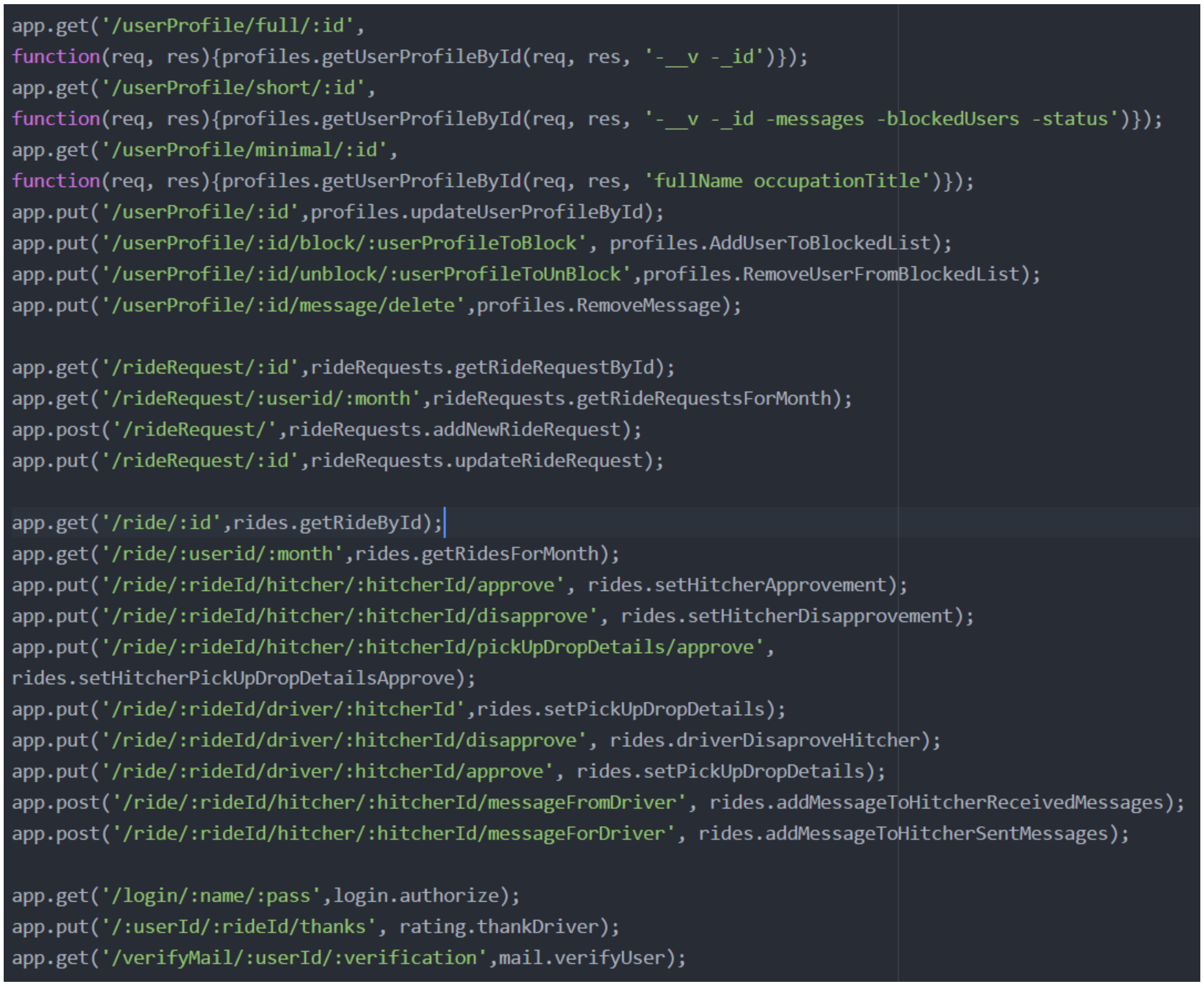
Node.js adds an enormous amount of new functionality to originally browser-based JavaScript. Primarily, the additions provide evented I/O libraries offering the developer system access, such as writing to the file system or opening another system process.

Node.js has very large and collaborative community that continuously improving existing technology and develop new libraries to expand the functionality. Those libraries are called packages in Node.js.

Web application framework

We use Express.js that is one of the most well-known packages for Node.js. It provides web application framework that is easy to configure. Moreover, Express.js provides a simple way for creating server REST API by defining routes. The next paragraph shows all routes of our web application server.

Routes



```
app.get('/userProfile/full/:id',
function(req, res){profiles.getUserProfileById(req, res, '-__v -_id')});
app.get('/userProfile/short/:id',
function(req, res){profiles.getUserProfileById(req, res, '-__v -_id -messages -blockedUsers -status')});
app.get('/userProfile/minimal/:id',
function(req, res){profiles.getUserProfileById(req, res, 'fullName occupationTitle')});
app.put('/userProfile/:id',profiles.updateUserProfileById);
app.put('/userProfile/:id/block/:userProfileToBlock', profiles.AddUserToBlockedList);
app.put('/userProfile/:id/unblock/:userProfileToUnBlock',profiles.RemoveUserFromBlockedList);
app.put('/userProfile/:id/message/delete',profiles.RemoveMessage);

app.get('/rideRequest/:id',rideRequests.getRideRequestById);
app.get('/rideRequest/:userid/:month',rideRequests.getRideRequestsForMonth);
app.post('/rideRequest/',rideRequests.addNewRideRequest);
app.put('/rideRequest/:id',rideRequests.updateRideRequest);

app.get('/ride/:id',rides.getRideById);
app.get('/ride/:userid/:month',rides.getRidesForMonth);
app.put('/ride/:rideId/hitcher/:hitcherId/approve', rides.setHitcherApprovement);
app.put('/ride/:rideId/hitcher/:hitcherId/disapprove', rides.setHitcherDisapprovement);
app.put('/ride/:rideId/hitcher/:hitcherId/pickUpDropDetails/approve',
rides.setHitcherPickUpDropDetailsApprove);
app.put('/ride/:rideId/driver/:hitcherId',rides.setPickUpDropDetails);
app.put('/ride/:rideId/driver/:hitcherId/disapprove', rides.driverDisaproveHitcher);
app.put('/ride/:rideId/driver/:hitcherId/approve', rides.setPickUpDropDetails);
app.post('/ride/:rideId/hitcher/:hitcherId/messageFromDriver', rides.addMessageToHitcherReceivedMessages);
app.post('/ride/:rideId/hitcher/:hitcherId/messageForDriver', rides.addMessageToHitcherSentMessages);

app.get('/login/:name/:pass',login.authorize);
app.put('/:userId/:rideId/thanks', rating.thankDriver);
app.get('/verifyMail/:userId/:verification',mail.verifyUser);
```

Node.js functions

Each function defined in route handles a part of application logic. When server gets a request for some URL, the appropriate function is called by web application framework with two main parameters – request and response. Function is responsible of doing some action with request according to application logic and returning a response on each situation.

Server-Client Response protocol

Node.js can return a response is all possible forms – as a text or JSON. To be consistent, we defined a protocol to be used by Node.js in response sending.

The protocol defines JSON structure:

```
{
	"status": STATUS,
	"message": MESSAGE,
	"data": JSON
}
```

Where

STATUS – string "success" in case of successful operation, "error" otherwise
MESSAGE – string – additional info from server. E.g. "Problem adding new ride request."
JSON – JSON object that is returned in response to GET request in case of "success" status.

MongoDB storage

We use MongoDB NoSQL database as our main storage.

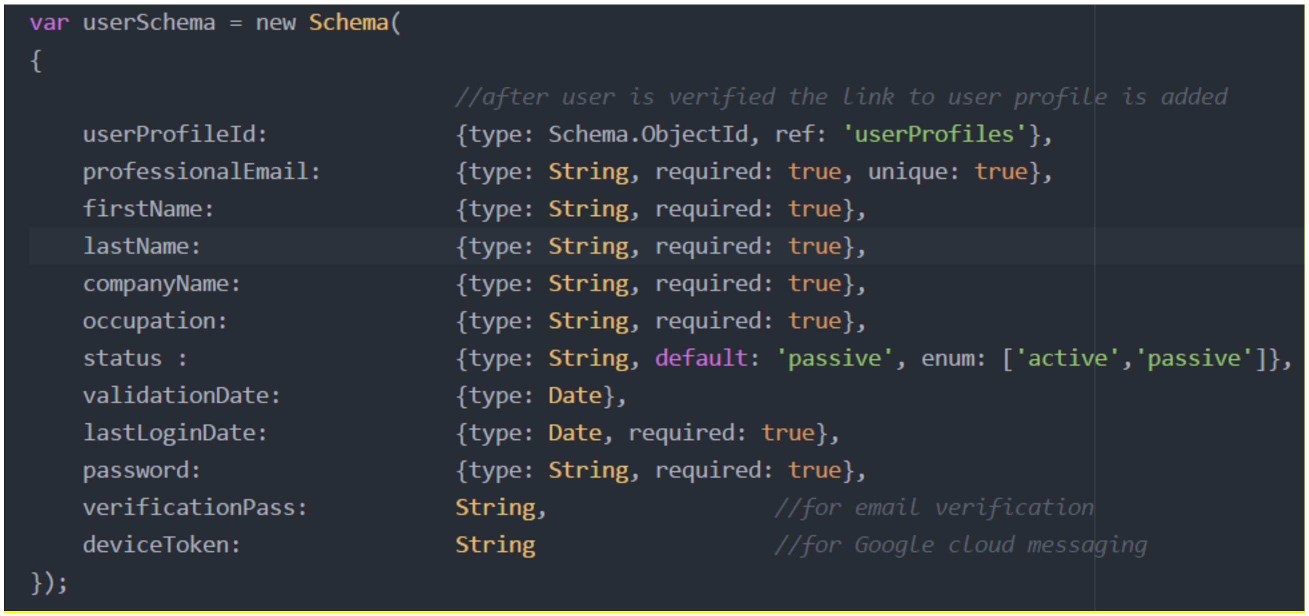
MongoDB technology

MongoDB is an open-source NoSQL database. It stores each record in a binary JSON (BSON) form, called document. Documents are saved in collections, like rows are saved in tables in RDBMS. MongoDB was built for horizontal scale using shards – collection parts, while sharding daemon will balance the load of the nodes in cluster and duplicate data to keep the system running in case of failure. Such maintaining data replication makes MongoDB highly available.

MongoDB is schema-less database, which gives us flexibility on the one hand, but requires to implement validation of schema on the application level on the other.

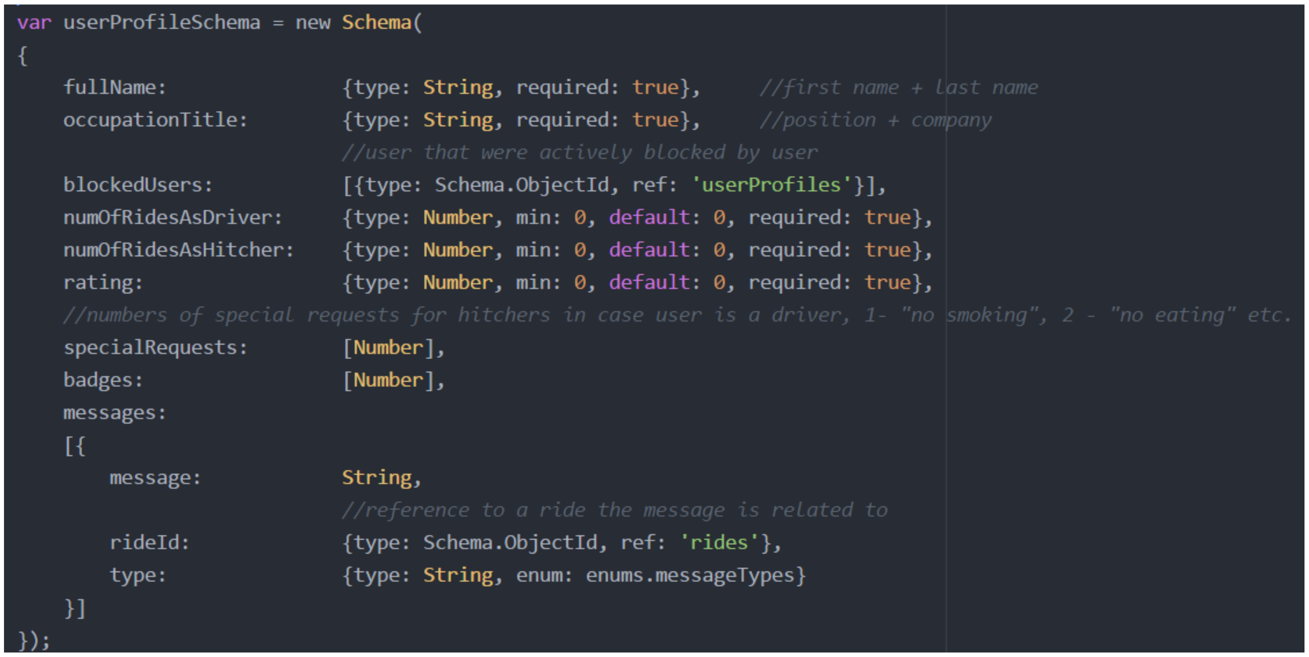
Mongoose Schema validation

In our project, we implement schema validation in Node.js with the help of Mongoose package. It requires schema definition and then exposes the model. Using mongoose model instead of direct mongoDB connection performs schema validation. Schemas are defined as JSON objects with definition of types and additional constrains to each field. Schemas of user, user profile, ride request and ride are shown below:



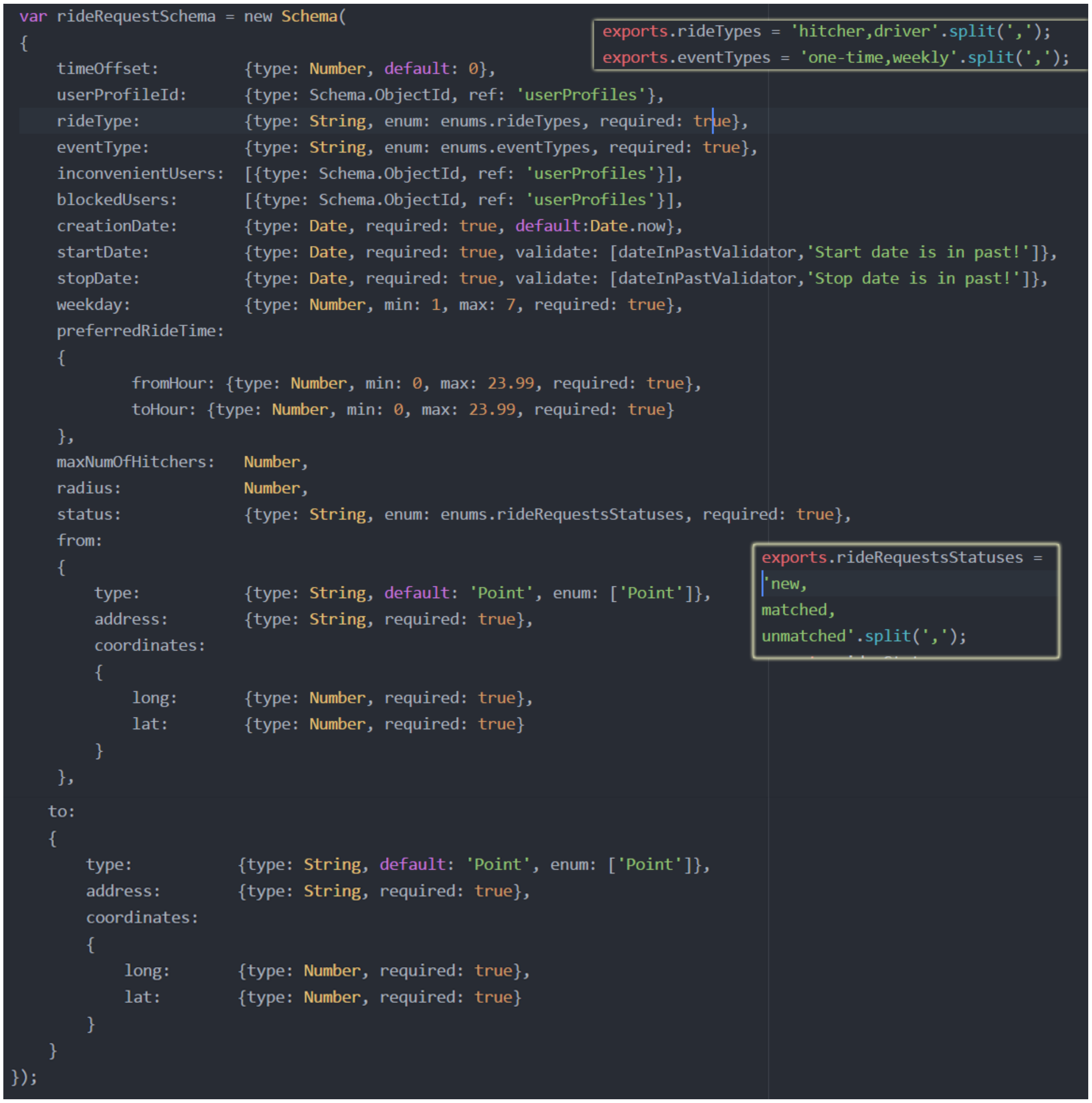
```
var userSchema = new Schema(
{
                                        //after user is verified the link to user profile is added
    userProfileId:              {type: Schema.ObjectId, ref: 'userProfiles'},
    professionalEmail:          {type: String, required: true, unique: true},
    firstName:                  {type: String, required: true},
    lastName:                   {type: String, required: true},
    companyName:                {type: String, required: true},
    occupation:                 {type: String, required: true},
    status :                    {type: String, default: 'passive', enum: ['active','passive']},
    validationDate:             {type: Date},
    lastLoginDate:              {type: Date, required: true},
    password:                   {type: String, required: true},
    verificationPass:           String,                 //for email verification
    deviceToken:                String                  //for Google cloud messaging
});
```

Figure 2 - User document schema



```
var userProfileSchema = new Schema(
{
    fullName:               {type: String, required: true},     //first name + last name
    occupationTitle:        {type: String, required: true},     //position + company
                            //user that were actively blocked by user
    blockedUsers:           [{type: Schema.ObjectId, ref: 'userProfiles'}],
    numOfRidesAsDriver:     {type: Number, min: 0, default: 0, required: true},
    numOfRidesAsHitcher:    {type: Number, min: 0, default: 0, required: true},
    rating:                 {type: Number, min: 0, default: 0, required: true},
    //numbers of special requests for hitchers in case user is a driver, 1- "no smoking", 2 - "no eating" etc.
    specialRequests:        [Number],
    badges:                 [Number],
    messages:
    [{
        message:            String,
                            //reference to a ride the message is related to
        rideId:             {type: Schema.ObjectId, ref: 'rides'},
        type:               {type: String, enum: enums.messageTypes}
    }]
});
```

Figure 3 - User profile document schema

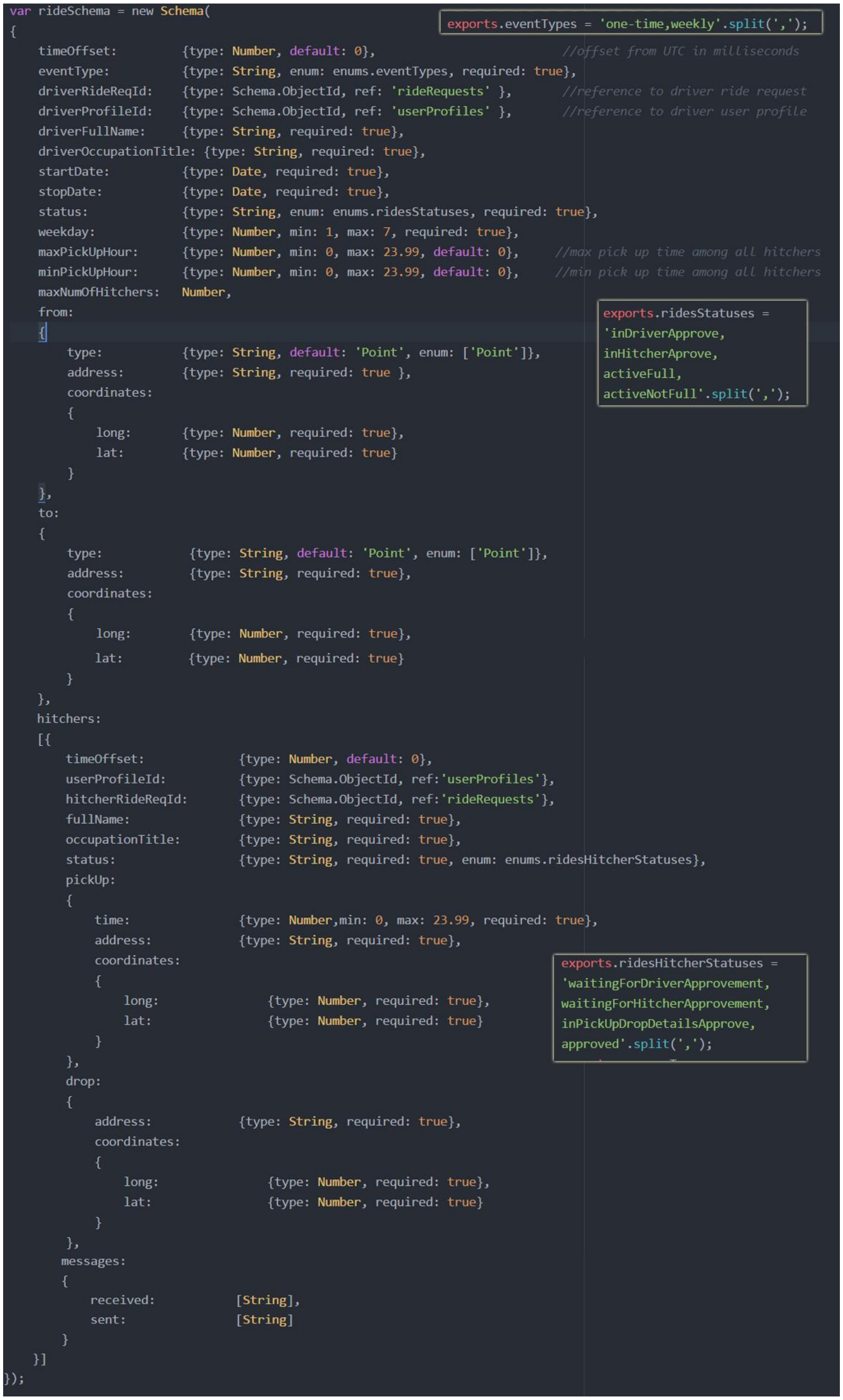


```
var rideRequestSchema = new Schema(
{
    timeOffset:          {type: Number, default: 0},
    userProfileId:       {type: Schema.ObjectId, ref: 'userProfiles'},
    rideType:            {type: String, enum: enums.rideTypes, required: true},
    eventType:           {type: String, enum: enums.eventTypes, required: true},
    inconvenientUsers:   [{type: Schema.ObjectId, ref: 'userProfiles'}],
    blockedUsers:        [{type: Schema.ObjectId, ref: 'userProfiles'}],
    creationDate:        {type: Date, required: true, default:Date.now},
    startDate:           {type: Date, required: true, validate: [dateInPastValidator,'Start date is in past!']},
    stopDate:            {type: Date, required: true, validate: [dateInPastValidator,'Stop date is in past!']},
    weekday:             {type: Number, min: 1, max: 7, required: true},
    preferredRideTime:
    {
            fromHour: {type: Number, min: 0, max: 23.99, required: true},
            toHour: {type: Number, min: 0, max: 23.99, required: true}
    },
    maxNumOfHitchers:    Number,
    radius:              Number,
    status:              {type: String, enum: enums.rideRequestsStatuses, required: true},
    from:
    {
        type:            {type: String, default: 'Point', enum: ['Point']},
        address:         {type: String, required: true},
        coordinates:
        {
            long:        {type: Number, required: true},
            lat:         {type: Number, required: true}
        }
    },
   to:
   {
       type:            {type: String, default: 'Point', enum: ['Point']},
       address:         {type: String, required: true},
       coordinates:
       {
           long:        {type: Number, required: true},
           lat:         {type: Number, required: true}
       }
   }
});
```

```
exports.rideTypes = 'hitcher,driver'.split(',');
exports.eventTypes = 'one-time,weekly'.split(',');
```

```
exports.rideRequestsStatuses =
'new,
matched,
unmatched'.split(',');
```

Figure 4 - Ride Request document schema



```
var rideSchema = new Schema(
{
    timeOffset:         {type: Number, default: 0},                               //offset from UTC in milliseconds
    eventType:          {type: String, enum: enums.eventTypes, required: true},
    driverRideReqId:    {type: Schema.ObjectId, ref: 'rideRequests' },         //reference to driver ride request
    driverProfileId:    {type: Schema.ObjectId, ref: 'userProfiles' },         //reference to driver user profile
    driverFullName:     {type: String, required: true},
    driverOccupationTitle: {type: String, required: true},
    startDate:          {type: Date, required: true},
    stopDate:           {type: Date, required: true},
    status:             {type: String, enum: enums.ridesStatuses, required: true},
    weekday:            {type: Number, min: 1, max: 7, required: true},
    maxPickUpHour:      {type: Number, min: 0, max: 23.99, default: 0},     //max pick up time among all hitchers
    minPickUpHour:      {type: Number, min: 0, max: 23.99, default: 0},     //min pick up time among all hitchers
    maxNumOfHitchers:   Number,
    from:
    {
        type:           {type: String, default: 'Point', enum: ['Point']},
        address:        {type: String, required: true },
        coordinates:
        {
            long:       {type: Number, required: true},
            lat:        {type: Number, required: true}
        }
    },
    to:
    {
        type:            {type: String, default: 'Point', enum: ['Point']},
        address:         {type: String, required: true},
        coordinates:
        {
            long:        {type: Number, required: true},
            lat:         {type: Number, required: true}
        }
    },
    hitchers:
    [{
        timeOffset:             {type: Number, default: 0},
        userProfileId:          {type: Schema.ObjectId, ref:'userProfiles'},
        hitcherRideReqId:       {type: Schema.ObjectId, ref:'rideRequests'},
        fullName:               {type: String, required: true},
        occupationTitle:        {type: String, required: true},
        status:                 {type: String, required: true, enum: enums.ridesHitcherStatuses},
        pickUp:
        {
            time:               {type: Number,min: 0, max: 23.99, required: true},
            address:            {type: String, required: true},
            coordinates:
            {
                long:               {type: Number, required: true},
                lat:                {type: Number, required: true}
            }
        },
        drop:
        {
            address:            {type: String, required: true},
            coordinates:
            {
                long:               {type: Number, required: true},
                lat:                {type: Number, required: true}
            }
        },
        messages:
        {
            received:           [String],
            sent:               [String]
        }
    }]
});
```

```
exports.eventTypes = 'one-time,weekly'.split(',');
```

```
exports.ridesStatuses =
'inDriverApprove,
inHitcherAprove,
activeFull,
activeNotFull'.split(',');
```

```
exports.ridesHitcherStatuses =
'waitingForDriverApprovement,
waitingForHitcherApprovement,
inPickUpDropDetailsApprove,
approved'.split(',');
```

Popular queries and information replication

Maybe reader have noticed that some information is replicated over documents in different collections in contradiction to well-known database normal form. The replication is done intentionally, according to the best schema design practices of NoSQL.

We need to look at the most frequent queries that will be performed in the system and place all the information related to that queries in one document. The main goal is to avoid join operations between documents – an overhead of network packages, web-server and database queues.

For example, one of the main and the most frequent operations of the system is viewing a ride. If the document was designed according to the normal form, the document would contain only ride related information and links to driver and hitchers. To show ride information to a driver we would need to get from server user profile documents of all hitchers and join them to the ride document. One additional server request for each hitcher. However, as we need this information very frequently (thousands of requests per day in working system), we can attach it to the document, and prevent server overload.

The information that is replicated changes very infrequently (once a year somebody got promoted or changes a workplace). In that case, we can pass over all rides and update the information. It's a small overhead once a year comparing to the thousands of events "saved" on a daily basis.

Indexes

MongoDB supports different types of indexes, which help to perform efficient queries. Some of them are created automatically, like index on "_id" field – the unique identifier of document in MongoDB, making queries findById very fast. Here is a table of all additional indexes we created to maintain our most popular queries faster.

Collection	Field	Type	Description
users	professionalEmail	Unique	Unique email identifies user for log in. In mongoDB index created to ensure uniqueness by rejecting duplicate values for the indexed field.
rideRequests	from.coordinates	2dsphere	All coordinates are indexed as geospatial spherical data. This index lies in a base of matching algorithm for searching nearest drivers' ride requests to match the hitcher ride request.
rideRequests	userProfile	Single field	The index makes possible efficient search for person's ride requests for calendar view.

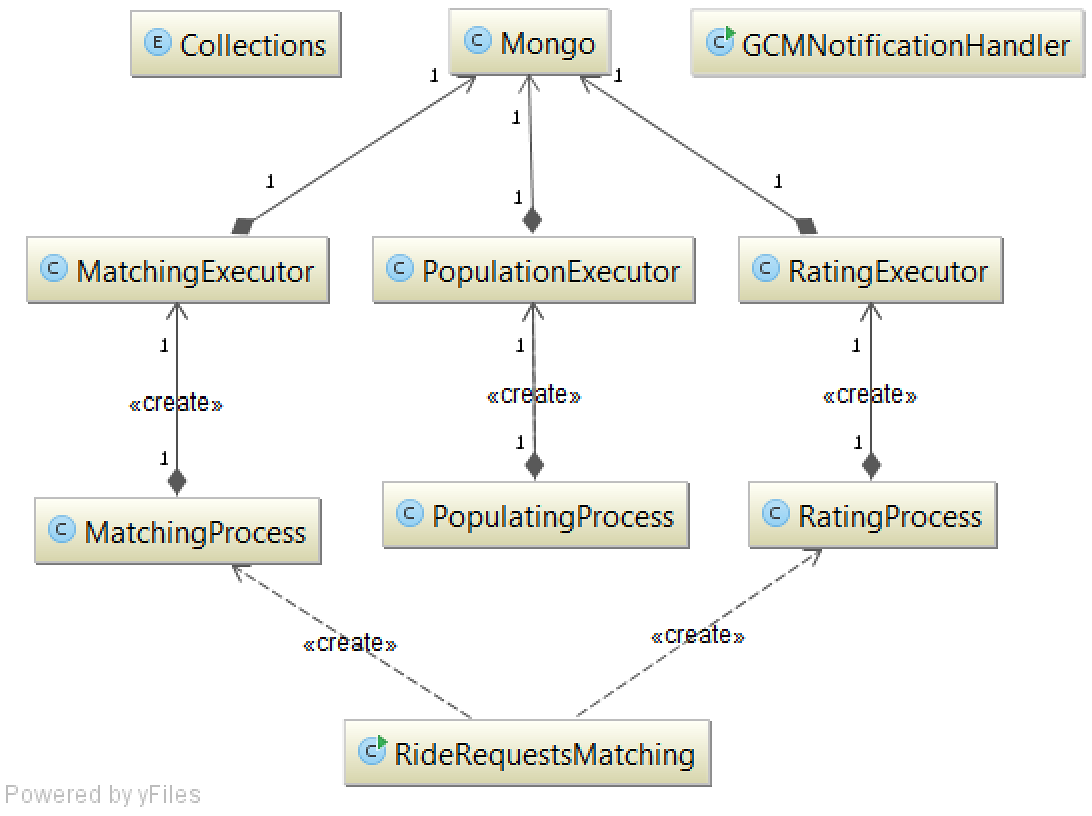
rides	from.coordinates	2dsphere	All coordinates are indexed as geospatial spherical data. This index lies in a base of matching algorithm for searching nearest rides to match the hitcher ride request.
rides	driverProfileId	Single field	The index makes possible efficient search of all rides where person is driver for calendar view.
rides	hitchers.userProfileId	Single field	The index makes possible efficient search of all rides where person is hitcher for calendar view.

Background server processes

Our project background processes implemented in Java program to run continuously on the server. For now main functionality - matching and rating – was added. In the future more processes, like reminder, can be added to program.

Java program design

The below diagram describes the structure of the Java program.



Mongo - singleton class responsible for communication to MongoDB database. It provides general methods for searching, updating and inserting document. Each of methods is synchronized to prevent the case two threads updating the same document.

Collections – enum, that defines collection names and contains collections for entire application usage.

Matching Process – runnable class that contains matching logic and run in a separate thread. It contains the main algorithm process, while all sorts of tasks in the algorithm are done with the help of Matching Executor.

Matching Executor – helper class for Matching Process class, contains a collection of methods that comprise matching process, e.g. find match for specific driver.

Rating Process – runnable class that contains rating logic and run in a separate thread. It contains the main rating process, while all sorts of tasks in the algorithm are done with the help of Rating Executor.

Rating Executor – helper class for Rating Process class, that contains a collection of methods that comprise rating process, e.g. send rating for hitchers.

Population Process – runnable class that populates batches of new random ride requests and run in a separate thread. It was used in an early stages to test matching algorithm. It contains the main populating process, while all sorts of tasks in the process are done with the help of Population Executor. In production, this class is not in use.

Population Executor – helper class for Population Process class, which contains a collection of methods that comprise populating process, e.g. Generate 200 new random ride requests. In production, this class is not in use.

RideRequestMatching – main class that starts Rating and Matching processes in a different threads, restart them in case of fail and gives them a command to stop when the time of running has elapsed.

GCMNotificationHandler – static class that is used to send notifications to Android devices using GCM Services.

Matching process

Matching algorithm always matches hitchers to drivers. Hitchers are divided to 3 groups:

- **reoccuringHitchersDocuments** - new weekly hitcher ride requests, algorithm tries to match them for the first time. This group has a priority, as our system tries to create long-term hitcher-driver relationship.
- **oneTimeHitchersDocuments** – new one-time hitcher ride requests, algorithm tries to match them for the first time.
- **unmatchedHitchersDocuments** - hitcher ride requests, that wasn't matched in a first or following matching loops.

Each ride request has statuses (see status field of the ride requests schema), which help to bring ride requests to all three queries in each matching loop.

Matching loop is the following:

```
while(!stopCommand)
{
    oneTimeHitchersDocuments = executor.getOneTimeHitchersDocuments();
    reoccuringHitchersDocuments = executor.getReoccuringHitchersDocuments();
    unmatchedHitchersDocuments = executor.getUnmatchedHitchersDocuments();

    for (DBObject doc : reoccuringHitchersDocuments)
    {
       matchFound = executor.matchHitcherRideRequestWithDriverRideRequests(doc);
       if (!matchFound) executor.matchHitcherRideRequestWithRides(doc);
    }

    for (DBObject doc : unmatchedHitchersDocuments)
    {
        matchFound =  executor.matchHitcherRideRequestWithDriverRideRequests(doc);
        if (!matchFound) executor.matchHitcherRideRequestWithRides(doc);
    }

    for (DBObject doc : oneTimeHitchersDocuments)
    {
       matchFound = executor.matchHitcherRideRequestWithDriverRideRequests(doc);
       if (!matchFound) executor.matchHitcherRideRequestWithRides(doc);
    }

    executor.emptyOneTimeHitchersDocuments();
    executor.emptyReoccuringHitchersDocuments();
    executor.emptyUnmatchedHitchersDocuments();
 }
```

Each loop brings new batch of hitcher ride requests that should be matched (to three groups). Then algorithms try to match each document consequently, first to new drivers, then to existing rides.

Search for drivers or rides is done with the help of MongoDB geosphere index, which makes search for nearest drivers very efficient.

```
DBCursor cursor = mongo.findDocuments(Collections.RIDEREQUESTS,
   new BasicDBObject("from.coordinates",
      new BasicDBObject("$near",
            new BasicDBObject("$geometry",
                  new BasicDBObject("type", "Point")
                     .append("coordinates", myLocation))
               .append("$maxDistance",  radius)
            )
      ).append("rideType", "driver").append("status", new BasicDBObject("$in", statuses))
   ).limit(MAX_NUM_OF_DRIVERS_TO_RETRIEVE);
```

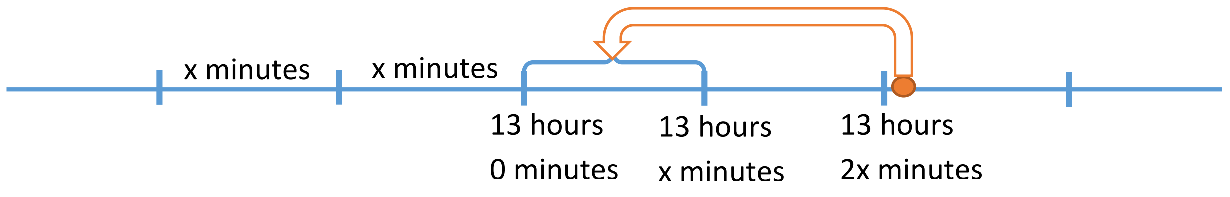
Queries' result is a finite number of drivers that start their trip in the radius specified by hitcher. The algorithm then checks if one of them match hitcher by destination distance, dates and time.

Rating process

In parallel with matching process, rating process runs in a separate thread. It detects which rides have just ended and performs the following steps to each ride:

- **Thanks messages distribution** - All hitchers of a ride got a message to thank a driver for a ride. By clicking the message, hitcher "says Thanks" and server adds +10 rating for driver.
- **Badges distribution –** server checks for each ride participant if he\she are ready to reveal a new badge. If so, the new badge is awarded to user and he/she got promoted in rating.
- **Ride messages removal** – all messages that were shared among ride participants were helpful at hitchers pick up. But are no longer relevant afterwards, so the messages are cleared to make room for new messages next week (for weekly rides).

Rating process looks at the time windows of predefined resolution and performs the above steps for rides that has ended in two time windows before the current time window.
For example let's assume the following timeline divided to periods of x minutes. The orange circle represents the current time point. At the red point the rating process will detect that time period has changed and will performs the above steps for rides that has ended at the time window 13:00 – 13:X.



Future Perspectives

CompanyRide application has big growing potential, here are some of the possible extensions:

- Make people benefit from growing rating – by getting discounts at some websites or by exchanging it to some valuable things.
- Participation in fuel expenses – drivers will be able to cover partially their expenses by sharing a car with hitcher.
- Similar to rating process that passes on all rides that have already happened, we can add reminder process that will send reminders to all rides that are about to happen.
- Now driver enters his starting point (from field) and application matches the hitchers according to this point. In the future this field may become an array of pick up points. With new functionality people will be able to pick hitchers on their way in different cities. This feature will increase the matching percentage and expose more drivers to hitchers and vice versa.
- Adding new features to application – like allowing people to ride only with people of his\her organization.
- LinkedIn connection and Facebook login.

Technologies

Server Side

mongoDB mongoose node JS Java

Client side

ANDROID Google cloud messaging Google maps

Development

Android Studio NetBeans TortoiseSVN IntelliJIDEA

Server instance

amazon web services™