Recent projects description

Nikolay Turpitko, Software Developer, Freelancer

authkit

https://github.com/letsrock-today/authkit

This project is half-finished, but the only one (of my projects) yet with a publicly available source code (all others are in private repositories, or an intellectual property of my clients or former employers).

We[1] started it to extract boilerplate auth code from the other our project[2] - "Let's Rock!"[3] (LR).

As LR project itself, we are developing authkit with a fellow developer[4].

Currently, for the authentication in the LR we use Nginx module, which we created using Go, `"golang.org/x/oauth2"` and `"github.com/dgrijalva/jwt-go"`. But, retrospecting it, we found, that we created more custom code than it deem to be necessary and it is a bit entangled.

However, it seems that we have a couple of useful pieces of code, which we would be glad to reuse in our future projects, and something might be even useful for other developers. Of course, we'd benefit if someone starts to use our code and contribute to it. So, we decided to publish it on Github under MIT license.

We have not finished to re-implement all the parts, we already had in LR, yet. It's not going so fast, as we hope initially, because we are trying to re-think our code, make it more simple and use as much ready libraries as possible. Also, we decided to do slightly more, than we already had in LR. For example, we blended Hydra[5] and Echo[6] into it.

So, we hope that when we reintegrate authkit back into LR, we'll have more secure, complete and, at the same time, more simple solution than we already have.

There is nothing much to say about this project. We made it open, everyone can check it out, skim through the README[7], see the code, execute demo using Docker Composer, follow it, give us a star, fork and experiment with it, create an issue or send us a PR.

Let's Rock!

https://letsrock.today

At the Google Play[8]

We[9] started this project along with colleague and a friend of mine[10] a couple of years ago. Our main goal was to learn Go programming language, Linux programming environment, and all the things, required for modern web programming, especially free and open source tools, techniques and practises - all that stuff, which we were discouraged to use during years of the work in the Java/enterprise environment.

This is an ongoing project, it is neither completely finished nor abandoned. We are doing it in our spare time, and it is not fully functional yet.

Sources are in the private repository. We hope to open it some day, but right now we are ashamed to do this. I may show some parts as a code sample by request. But the whole repository contains some sensitive parts. However, we started to extract authentication code from this project into open-source one, see authkit[11].

[1] https://github.com/letsrock-today
[2] ../lr
[3] https://letsrock.today
[4] https://ru.linkedin.com/in/ustinovandrey
[5] https://github.com/ory-am/hydra
[6] https://github.com/labstack/echo
[7] https://github.com/letsrock-today/authkit/blob/master/README.md
[8] https://play.google.com/store/apps/details?id=today.letsrock.client
[9] https://github.com/letsrock-today
[10] https://ru.linkedin.com/in/ustinovandrey
[11] ../authkit

Basically, our server uses public API of several ticket sellers (similar to SeatGeek[12]) to retrieve information about entertainment events into our database, index it and provide better search and filtering, as well as additional features, like push notifications about new events, conforming user-defined criteria. We also planned some social aspects, but haven't implemented these ideas yet. Another planned, but not implemented yet feature is to enrich information about events from the public media (Youtube, Wikipedia, etc). So, basically, our app is an aggregator. But we are trying to provide some additional services to the end user, and not pretend to be a reseller. We do not (and not going to) sell anything within our app, we forward traffic back to the partner site (with referral code, of course). This model is not going to be very lucrative, but it's OK for start.

So far we created:

- back-end server on Go,
- very basic web app (HTML5, Riot.js[13]),
- more functional native Android app,
- build and deployment automation scripts for the server.

Server

We use Go-based services (we call them "modules") behind the Nginx[14] server, working in the reverse proxy mode. Each service focused on one task, has its own database and API. Architecture is not strictly "micro-service", though, because services are relatively coarse-grained and, moreover, we allowed them to share databases in some cases. For example, we have services which populates data into the DB, and others that perform search and returns data to the client app via API, they necessarily share the same databases.

We use Redis[15] and LedisDB[16] (with LevelDB[17] back-end) databases, plus Sphinx[18] search engine. Redis we use for small frequently used objects (because Redis stores everything in RAM), LedisDB - for all other things (which are stored mostly on HDD, but with mmap[19] for faster read access), and Sphinx stores indexes mostly. Each service has several logical connections (we call them "data sources") to the databases it needs. We are striving to logically separate data sources, though, currently they may reside within the same DB instance, and some services may use the same logical data source by necessity.

We think, that with the current architecture we are well prepared to scale, if necessary. Actually, we haven't proved this in practice yet. Currently we deploy all the components onto the single virtual host with unix-socket connections between them. But, theoretically, we can deploy Nginx, (multiple instances of) every service and database (cluster) onto the separate nodes, using Nginx as a load balancer.

We performed some stress tests on our current setup and monitored CPU, RAM and HDD consumption growth and estimated, that we can grow up to 10K simultaneous connections with only VDS hosting tariff plan upgrades, after that we have an option to deploy databases onto separate nodes, which can give us some time to prepare for the farther traffic growth. Still, it's all pure theoretical, actually we'd be quite happy to become popular enought to have a small "slashdot effect".

We used Gorilla toolkit[20] and Negroni middleware[21] to implement services. Each service deployed as a separate RPM package[22] (we use CentOS Linux[23] in production). Currently, to simplify deployment and version control, we use single super-package, containing only common initialization scripts and dependencies onto other app's packages. However, we can deploy packages independently almost without interruption. Services are completely stateless, behind the load balancer and we use graceful shutdown to minimize loss of active connections.

We use systemd[24] to manage our services. It capable to detect if processes are stuck or dead and restart them automatically, if needed. We also used systemd's socket activation - mode with which client connects to the socket, managed by systemd, so, socket would not break if process crashed and restarted. We leveraged systemd's watchdog and have quite a few of systemd timers.

One of the server's modules holds information about users. It also used as a request handler of Nginx's http auth

[12] https://seatgeek.com/
[13] riotjs.com/
[14] https://nginx.org/en/
[15] https://redis.io/
[16] http://ledisdb.com/
[17] http://leveldb.org/
[18] http://sphinxsearch.com/
[19] https://en.wikipedia.org/wiki/Memory-mapped_file
[20] www.gorillatoolkit.org/
[21] https://github.com/urfave/negroni
[22] http://rpm.org/
[23] https://www.centos.org/
[24] https://www.freedesktop.org/wiki/Software/systemd/

request module[25]. It implements OAuth2 3-legged flow with "golang.org/x/oauth2"[26]. We started to re-write it as an open-source project (see authkit[27]).

Build and deployment automation

We started to implement our build automation logic when Docker[28] was young, and Docker Compose[29] was to be born yet, so, we created relatively complex make build to manage docker containers. We use docker containers to:

- build binary files from Go sources for dev-test (and optionally execute unit tests and lint),
- setup dev-test environment (to perform integration tests - with databases, Nginx, Sphinx, etc),
- build RPMs (unit tests and lint run again as a part of this build).

After RPM files are created with docker, make executes Packer and Vagrant to prepare fresh test environment within virtual box, in which it executes tests of the deployment, undeployment and some integration tests. This environment closer resembles production, and allows us to find some issues which Docker doesn't. For example, that's how we test required permissions, selinux rules, etc.

We also use make and bash scripts, to prepare production server and to deploy (or update) server application into production. No Puppet or Ansible at this point, just don't need them yet.

Native Android app

For this project we decided to use native Android app. I have had awful experiences with HTML5 / Apache Cordova app in Taxi project[30].

In our Android application we used following libraries:

- Android Annotations[31],
- Volley[32],
- CardView and RecyclerView[33] from support library,
- FireBase[34] for push notifications.

To support custom OAuth2 login, we implemented Authenticator Service[35] and used WebViewClient to perform 3-legged flow (redirect user to the provider's login page).

Web app

There is not much to say about the web app. It is HTML5 single-page app, created with Riot.js[36]. We used es6 (babel)[37], npm[38], webpack[39] and i18next[40].

GetStar

http://getstar.net

Sources are in the private repository.

This project we made in the tight collaboration[41] with my buddy (and a former colleague of mine)[42].

[25] http://nginx.org/en/docs/http/ngx_http_auth_request_module.html
[26] https://godoc.org/golang.org/x/oauth2
[27] ../authkit
[28] https://www.docker.com/
[29] https://docs.docker.com/compose/
[30] ../taxi
[31] http://androidannotations.org/
[32] https://developer.android.com/training/volley/index.html
[33] https://developer.android.com/training/material/lists-cards.html
[34] https://firebase.google.com/
[35] https://developer.android.com/training/id-auth/custom_auth.html
[36] WebViewClient
[37] https://babeljs.io/docs/learn-es2015/
[38] https://www.npmjs.com/
[39] https://webpack.github.io/
[40] i18next.com/
[41] https://bitbucket.org/spooning/
[42] https://ru.linkedin.com/in/ustinovandrey

Project was about creating a toolkit for near-telecom startup to analyze telephone traffic with aim to detect suspicious activities and to alert staff about them.

Task was somewhat experimental. Our customer required some degree of flexibility out of the tool, because the most suitable algorithm for the task was unknown and to be found. So, the tool should be configurable and convenient to experiment with different algorithms and parameters. Also, it was intended to be used in two different modes - for a research and analyzing and as an alerting service. Customer stated that they would appreciate timely developed and deployed tool and prefer fast, maintainable and extensible solution over GUI.

Because of that, we decided to create the set of tools in a more Unix way. We developed several tools for specific sub-tasks which could be combined into one service using a bit of a Bash scripting. Tasks could be executed manually and individually as well as in the single pipeline as a periodic service, controlled by systemd.

In Java world we would use Apache Camel[43] for similar task, but we decided, that Java is an overkill in this case. Java itself requires regular maintenance and have too heavy system requirements. We decided to use Go and standard Linux tools, which adds small overhead to the PostreSQL installed on the same virtual machine.

We created several tools which:

- download CDR[44] files from (potentially) different sources (web/ftp),
- parse file formats of particular phone operator and load data into DB (PostgreSQL),
- process data, using configured algorithm and invokes post-processing to make reports, send mails about suspicious conditions or execute configured script,
- display plots for analysis in the simple web UI.

Nothing particularly fancy here. Some interesting points:

Where possible, we used existing Linux utilities, executing them from Go program as external process, if necessary. Go program reads custom configuration and adds custom logic to the tool. For example, we used `lftp` to download files from ftp, `find` and `sort` to prepare lists of files to process, `tar` to archive, `iconv` to convert encoding, `sed` and `column` to prepare simple reports, `mutt` to send emails, etc.

We used `yaml` configuration files and allowed putting `bash` scriptlets within it. Tool would populate environment from configuration and current application state and execute bash scriplets for particular sub-task. This way we allowed system administrator to use familiar tool to extend application. Solution is flexible enough, it allows executing arbitrary script or program for the task. If standard Linux utilities won't be enough, it's possible to create script on `awk` or any scripting language, or even to execute binary program, created for the task later. Solution is safe enough (through right file permissions and specially created user to execute service).

We developed several simple algorithms to calculate traffic forecasts and some more elaborated (combined from simple ones) and provided an extension point - ability to execute external process for calculation. Several external processes could be started to process data in parallel, each process started only once and live till the whole calculation is done. Every process reads calculation tasks from `stdin` and returns individual results to `stdout` (we used simple custom binary protocol based on `msgpack`). Go program than send results via channel to the next stage (post-processing), where individual results are accumulated and then similar approach used to concurrently execute post-processors (which again could be an external scripts).

We used `"go/types".Eval()` to dynamically execute conditional expressions, which can be stored in the configuration file in Go syntax. These expressions are evaluated over fields of report and allow filtering data for reports and changing of alerting conditions without re-compilation of the whole program.

We created prototype and demonstrated ability to execute more complex external algorithms (in different execution environment), using provided extension point. We executed R-SSA[45] (which is implementation of SSA[46] algorithm, written on R programming language) to create traffic forecasts, feed it with data from Go program and captured results back into Go program.

It's may be interesting to note, that we achieve acceptable performance with simple external algorithms, even written on scripting language (again, we tried R for simple tests). With such complex algorithms as R-SSA, performance degraded considerably. R-SSA with large "window" ate all CPU cycles (and with short "window" it gave unreliable predictions). Calculations could be done on real Core i7 notebook, but were unacceptably sluggish on the provided 1-CPU virtual machine. We stopped experiments due restrictions of time and budged. Still, I think potentially we would squeeze more interesting results with a proper setup (it would be interesting to execute math-expensive calculation on the simple cluster of 3-5 dedicated R nodes, or try to develop our own SSA implementation on compiled language). Nevertheless, I think, we gave a good tool to the customer's engineers. They can experiment, explore and extend it as they need.

[43] http://camel.apache.org/
[44] https://en.wikipedia.org/wiki/Call_detail_record
[45] https://cran.r-project.org/web/packages/Rssa/index.html
[46] https://en.wikipedia.org/wiki/Singular_spectrum_analysis

We used systemd to set up periodical execution of our alerting service, to start web service on startup and to implement error monitoring. Error monitoring tool was a simple script, which is periodically executed by systemd. It simply executes journalctl to find service's errors in the system journal for the last hour, and send email, if there are such errors. Much simpler, than we used to do it in Java.

We used `make` to build all binaries from Go sources, to prepare `man` documentation and to prepare deployment. `go build` alone was not enough to all these tasks. We used `pandoc` to create `man` pages from Markdown files. We used `dh_make` and `debuild` to create Debian deployment package.

Overall, we were pleased with Go for this task (and with our collaboration with customer, looking forward to work with them again).

Taxi Moment

http://taximoment.ru/

Sources are in the private repository.

This was a project of my former colleague[47], who asked me to help him with Android application for booking taxi cabs in Moscow.

I implemented:

- web/mobile app for booking taxi cabs (not the whole website above, only part for user to book a cab),
- back-end service for this web/mobile app,
- prototype of mobile app for drivers (didn't make it into production).

Web/Android/iOS app with Apache Cordova, jQuery and Dojo

Here is how it looks in the browser[48] and at the Google Play[49].

We decided to implement it using relatively new (at that time) HTML5 and Apache Cordova[50]. This way we hoped to use the same HTML and JavaScript code for web application (deployed on website) as well as for the Android and iOS apps.

Initially I tried to develop it as a single-page jQuery[51] app, but encountered severe performance issues within Cordova app on all devices I used to test it at that time (an old Motorola Android phone, presented to me by another my former colleague when he heard that I'm going to learn mobile programming, relatively new Android tablet, proudly made in PRC, iPhone 3, and all emulators).

So, to be able to at least test it (needless to say about customers, they, strangely had more modern and powerful devices) I decided to switch to another, more lightweight JavaScript framework. I evaluated several of them, popular at that time, and chose Dojo[52]. It looked solid, mature, and offered a compiler, which optimized, minified and packaged only reachable JavaScript code (there are plenty of alternatives nowadays).

Reimplemented with Dojo, application became significantly lighter, it stopped to crash on Motorola now and then. But I was still disappointed at Apache Cordova performance and development clumsiness. Don't know, if it's became better since then, but I decided **never touch Apache Cordova (or similar products) again**.

It turns out, that we spent more time patching and plumbing our app, making the same codebase to work at completely different environments, then we, probably, would spend, developing native versions for every platform. For such a small and simple app it does not worth it. And for bigger app, I suspect, the amount of differences is much bigger, so plumbing will take even more efforts.

Another aspect with such approach is that from time to time you have to implement some native code anyway. You have to create plugins. For example, when I developed Android app for drivers, we needed it to work in background, updating driver's GPS position at taxi operator's servers. And it was not possible without help of native code. I created plugin for it on Java for Android. OK. But it was broken Cordova's promise. Code base was no longer the same for different platforms, we had to support not only "common" JavaScript code, but also native plugins. And moreover, user experience was not the same as with native apps. I was not able to explain to every Android and iOS

[47] https://ua.linkedin.com/in/alexander-larionov-61536764/en
[48] http://taximoment.ru/mobile/?lang=ru&theme=Custom&sprut-url=http://momenttaxi1.appspot.com
[49] https://play.google.com/store/apps/details?id=ru.taximoment.cab
[50] https://cordova.apache.org/
[51] http://jquerymobile.com/
[52] https://dojotoolkit.org

user why UI is so ugly and different from native. Why, looking almost as native (but for obsolete phone versions), it behaves differently in some unexpected ways.

Nevertheless, application was created, and used to work on both iOS and Android. We stopped to develop it due shortage of funds from business side, but from time to time it requires maintenance (again, due Apache Cordova security breaches and updates). Recently we decided to drop maintenance of iOS version, just because of that. It was already expensive enough to pay for App Developer account every year just for the single application, which does not generate much income. But we were not prepared to fix application after every Apache Cordova update, which came out of our schedule, and was not welcomed by our business customers.

Besides Apache Cordova, jQuery and Dojo, I had a somewhat bitter experience with Google Maps and Geolocation API. It worked. We used it to show booked route on the map, to estimate price, to self-check by the user that he/she entered correct addresses. Most of the times it worked well. But there were errors, when it found entirely different geo-position for address, provided by user. User was disappointed. We could do nothing, that's how Google parsed entered address, probably, because correct address was not in their database. We could not solve this at our side. Human operator during phone call usually corrects this issues, but blames us at the same time. Another issue with Google APIs is that Google likes to drop its products. We used GWT in admin's web app (it was not created by me, but I helped to fix several issues in it). When Google dropped GWT support and changed maps or geo API, we was in unpleasant situation.

Server module for client auth, and update order status (Go, Google App Engine)

This part was fun. When I almost finished client app's, I needed to add simple login into app, so that returned user could check status of his/her order(s) and could collect bonuses for loyalty. It needed new and relatively independent service at server side. So, I decided to implement it on Go programming language. Just for fun and to try it in some project. We already used Java on Google App Engine for the rest of the server app's logic. App Engine offered Java, Python and Go. I was fed up with Java and had some brief experience with Python, so I decided to try Go, and was charmed with its simplicity, clarity and performance. I mean not execution performance. All we know, that Java is pretty performant when it is wormed-up. But I mean development performance in the write-compile-execute loop. How fast it compiled and started to work. I was not able to make a cap of tee, relax and forget what I was about to do during build, like I used to with Java. That how I started to like Go.

For the app I implemented OAuth2 3-legged authentication (social login). App did not have hard security requirements. It was enough to login user with his/her social account (Facebook, Google, Twitter) and simply check next time that it is the same user. I used "code.google.com/p/goauth2/oauth" for this (and "github.com/mrjones/oauth" for Twitter).

As one of the options besides social login, application allows entering app using phone number and one-time generated pin code sent via SMS. I connected app to Twillio and SMSAero providers for this task.

Main server pushed order statuses to the service, service saved them into database and returned them on requests from client app. JSON format was used for messages, appengine's datastore and memcache API were used for persistence.

Additionally, service provided simple address book for client's regular destinations and made Google Geocoding API requests to parse address for address book. Also, it proxied some requests to 3-party API to find street names by first letters (for suggestion box in the client's app).

Nothing fancy. It just works unattended for years.