Part-up Architecture

After a successful concept and designing phase with Lifely and Part-up, it is time to decide upon the approach we will be taking to turn the Part-up concept into a minimum viable product. In this document we will explain and argue our choices for the technology / software stack and the development methods.

Premises

- The first version of the application should be deployed and open for end users before the end of july. This is a short timeframe (as of today 20-02)
- The project should be easily accessible for contributors
- The success of the concept relies heavily on collaborative Part-up creation

Scope

The first iteration of the Part-up mvp roughly consists of the following big points

- user acquisition flow
- Part-up entity management (Part-ups, activities, contributions)
- activity feeds
- anti-contracts (Part-up “snapshotting” and “signing”)

The elaborate suggestion engines and graph / data analysis are not yet part of this version but will be taken into account in the choice of architecture.

Application development with Meteor.js

For this project we want to use the Meteor.js open source full stack javascript framework. Meteor.js is built on top of node.js and mongodb and exposes its data through ddp sockets to the client. For the client application we will use blaze, Meteor’s reactive frontend framework.

development speed

Developing a web application in Meteor is extremely fast due to all of the functionality that is provided by the framework. We have got a lot of previous experience with RESTful API’s and AngularJS frontend but find that a lot of time is lost in defining and integrating API’s between backend and frontend that we could have spent building or changing features in the product.

In the long run, Part-up will need a REST API to allow integration with 3rd parties and the creation of a developer ecosystem. Because we are not building the REST API from the get-go we are taking this technical debt to deliver the MVP earlier.

real-time reactivity

Because of the reactive data model in meteor applications and the socket connection to the backend the built interfaces are real-time by definition. We believe that the future of the web is realtime and reactive because end users are used to these kind of features from web applications that they use everyday.

Meteor is not the only reactive framework available but is a very complete one that does a good job at it. Meteor uses open source libraries such as SockJS to handle the fallback when websockets are not available.

The alternative to reactivity with Meteor would be to build a REST API with a separate streaming API on top of it, which would use a library like SockJS or socket.io to supply the streaming features.

frontend translatability / internationalisation

The management of labels and translation of labels in the application will be done using the messageformat library (http://messageformat.meteor.com/examples). This allows us to both manage the original language labels and switch out the labels to another language when needed. Messageformat also allows us to use templates to translate texts with variables in them.

Datestamps will be stored in Zulu time format. Data stored in mongo will be default UTF-8.

In the long run, when scaling up to multiple languages and locales, using a solution with message formats become increasingly difficult to maintain, especially when designing the interface for the asian markets and “Right-to-left”.

The alternative would be to construct different frontend templates for each of the required languages, as done with ING. For now we take the technical debt to get the dutch and english version delivered with the messageformat as fast as possible.

version control

For Part-up’s version control we will be using git, hosted at bitbucket.org. We follow the git-flow (http://nvie.com/posts/a-successful-git-branching-model/) branching model, which means we have a master branch containing the production release code. The develop branch contains finished but unrelease features, that are developed in separate feature branches until they are ready for deployment and merged back into the develop branch.

Testing, Continuous Integration and Deployment using Wercker

We will be using the velocity test framework to perform unit tests and integration tests on the Meteor application. All branches will be continuously integrated through wercker.com that spins up clean boxes running the tests. All successful builds on the develop branch will be automatically deployed to a staging environment.

We will be creating separate staging, acceptance and production environments. Committed code will automatically be deployed to staging and release candidates will be pushed to acceptance to allow QA before pushing the code to production servers.

Hosting and scaling with modulus.io

We've got a lot of experience with deploying applications to our own digitalocean boxes provisioned with ansible scripts. However, for meteor we will be using modulus.io to acquire advanced scaling options and features such as "sticky sessions" (see below) without completely setting up this infrastructure ourselves. In the long run, this will be an option but for the current scale of the project this solution is perfectly suitable.

Scaling topics

"Sticky sessions", or "session affinity" in meteor terms means a user that is using longpolling fallbacks will follow the same route through the load balancer. If the user is connected to a different machine, the state of the database cache will be lost and the data will be sent again. This has nothing to do with the login state, as session tokens are used that are persisted to the mongo database. (http://docs.meteor.com/#/full/accounts_api)

Micro services and inter service communication

The Meteor stack is very suitable to quickly implement the presentation layer of a web-application. The data is exposed real-time through the oplog of the mongo database. This does not mean we have to use the meteor app to do everything in the application.

Other parts of the system such as the Part-up / Upper recommendation engine will be split up into separate microservices. Communication between these microservices will be achieved through an event sourcing mechanism in which Meteor will produce events that are published on a pub/sub bus such as RabbitMQ and recorded into an event log.

A microservice such as the actor network taking care of the recommendations can very easily hook into the events published on the bus and stored in the event log. This would completely separate the storage of the frontend database and the storage of the connected microservices. The events produced by the actor network would be fed into the event source log where the Meteor app would subscribe to update the frontend database.

(future) A/B testing

There are several possibilities for doing A/B testing in Meteor, either internally by using the mongodb to store the results or integrating a hosted third party solution like Optimizely.

(future) Suggestion engine

For the future suggestion engine we'd suggest using Neo4j to build the first version of a graph based engine that is able to do construct a graph based on the event source log. Queries operating on this graph would produce interesting related entities through relations. (https://www.youtube.com/watch?v=qbZ_Q-YnHYo).

When reaching the limits of Neo4j on a single box during scaling, it would make sense to switch technology and use an actor framework such as scala with akka to scale the access to the recommendations.