The TransferWise Stack— Heartbeat of our little revolution

As any tech startup that's passed its five-year mark, TransferWise has come quite a way from the first lines of code that powered it. Our product is a living organism, with changing needs. What was right yesterday isn't necessarily so today. Nor might our current technology choices withstand the test of time. We hope they do - but maybe they don't. And it's okay.

At conferences and meetups people often walk up to our engineers to ask what languages, tools and technologies we're using. Fairly so, as we haven't done a stellar job of telling our tech-savvier customers and fellow developers much about that. Hopefully we can rectify that a bit by taking time now to reflect in writing.

We'd love to hear back from you about the decisions you would have made differently.

Brief history

Once upon a time, in 2010, there was a founder who wanted to solve a problem. He knew a bit of Java and wanted to get moving quick. At that time, Groovy and Grails seemed to have brought some of the Ruby on Rails flare to the JVM world. Boom, here's a quick way to bootstrap! By end of 2013, about a dozen engineers were working on the codebase.

In early 2014, the existing Grails-powered system wasn't cutting it anymore for some workloads. It had been quick and easy to deliver new features but the team had made some system design shortcuts on the way. The time had come to extract some critical batch processing into a separate component. We've been following the path of moving code out ever since.

By late 2016, TransferWise has about 120 engineers working in two dozen teams. Measured by lines of code, more than half our business logic lives in separate services. We're looking at ways to scale the system across data centers and coping with the next 80 engineers joining during 2017. Let's get into the details of how we intend to enable these moves.

Microservices - Spring Boot and Spring Cloud

Few contenders got to the starting line when picking between the possible groundworks for our service stack. We were sure to stay on the JVM. We wanted something that would promise good support for future "cloud" concerns. These included service discovery, centralized config and transaction tracing.

Many people in the team trust the quality of thinking going into the Spring ecosystem, so Spring Boot quickly gained popularity. It provides a good, extensible platform for building and integrating services. We like its annotation-based autowiring of dependencies, YAML configuration files and reasonable conventions. We use a custom Spring Initializr to bootstrap new services. That helps us to make sure all the needed bootstrap config is in place and nobody gets a headache trying to manually bring in the right dependencies. It's all running on Java 8.

Spring Cloud adds integration for many useful tools in a service-oriented environment. Some are Spring projects, like the Config Server. Some leverage battle tested open source components like Netflix Eureka, Zuul and Hystrix.

We are actively adopting Config Server and Eureka, and use Hystrix and Zuul in appropriate places.

Grails and Groovy

Grails and Groovy currently power all our business logic that's not extracted into microservices. That makes up a bit under half of our lines of code. We've taken a clear direction to deprecate this codebase by end of next year.

When Groovy came along, it brought with itself a chance to write nice, succinct code and leverage the whole Java ecosystem. It continues to be a good language for DSL-s, for instance. Groovy used to have more benefits over Java, like functional constructs, easy collection processing and other boilerplate-reducing tidbits. Java gives us better compile-time checking and less runtime overhead. Hence we've gone with Java in our micro-services.

Neither is Grails to blame for anything. It allowed the early team to quickly iterate on a product in its infancy. The convenience features of Grails served against it over the years. Grails hides complexity away from the developer. It makes it easier to shoot oneself in the foot when trying to deliver value. By taking a decision to focus on scalability of the codebase sooner, we would have been able to postpone migration by another year or so. Yet, in our opinion, Grails makes sense as a platform for a single moderately-sized web app - rather than for dozens of microservices. This made the transition inevitable in any case.

It's worth noting that latest Grails version, 3.x is, also, built on top of Spring Boot. As we're quite happy with plain Spring Boot, we are not currently considering it.

Persistence layer - MySQL, PostgreSQL

We're a financial service company. This instructs us to always prioritise consistency over availability. BASE is fun and eventual consistency sounds like a cool topic to wrap one's head around. We want our persistence to meet the ACID criteria - Atomic, Consistent, Isolated and Durable.

TransferWise started with MySQL. The widespread adoption, ease of setup and loads of people with some basic experience made it a good choice. With growth, more questions have come up about our DB engine, like:

- does it support our analytical workloads?
- is it easy to build High Availability?

MySQL still holds most of our data in production. Yet, migrating our analytical datastore to PostgreSQL is already underway as our tests show it to be a better fit for our models and tooling. Our financial crime team relies on many nifty PostgreSQL features. We foresee Postgres to also be the platform of choice for our future service datastores. Mature availability and resilience features it offers out of the box drives this. We like Postgres.

It's likely that we'll be adopting NoSQL for some use cases down the road, where referential integrity doesn't add value.

Messaging & Kafka

A big part of our business logic swirls around the state changes of a transfer. There's many different things that need to happen around the changes - like fraud checks and analytics updates. Earlier, these were all done synchronously together with the state change.

As the business has grown in complexity, that obvious and simple approach doesn't scale so well. A good rule of thumb in designing data flows is that we can make every signal that doesn't

need an immediate response asynchronous. If you can take a document to another department and not wait around for them to process it, you can process it asynchronously in the code as well. We want to unbundle the reactions to an event from the transactional changes of an event. That makes it easier to scale the different parts and isolate potential problems.

In the first iteration of our payout handling, we experimented with Artemis as a messaging platform. We didn't become confident about running Artemis in a HA setup in production, and now most of our messaging has moved to Apache Kafka.

The front end

That's all fine, but Kafka doesn't let you create smooth user experiences!

In 2014 TransferWise web UIs still used good old jQuery. We were not happy with the testability of that codebase. We also knew we'd need to modularize in a while due to team growth. In September that year, we launched our revamped transfer setup page built using AngularJS. We've now adopted Angular for almost all parts of our website.

We use Bower and NPM for dependency management. Grunt automates the builds. Jasmine and Protractor power the tests and Karma + PhantomJS run the tests. There's also webpack and Babel doing their part. Some parts of the code already use TypeScript. Whew.

Front-end, of course, isn't only code. Practices matter more than using the latest-and-greatest tools.

The teams in TransferWise are quite independent in how they evolve the product in their areas of ownership. This has, at times, meant trouble governing the visual identity and making sure things look neat, clean and consistent across the site.

To help with that we've taken a battle tested approach of implementing our style guide on top of Bootstrap. And, for the record, we think that LESS is more.

We believe in TDD, which is particularly important in places with less compile-time safety - like Javascript. Ürgo has blogged before about some of the thinking we apply. There's more to share. Stay tuned for future posts covering different aspects of our experience in detail.

Running in production

So, how do we keep it together? How do we make sure our systems are serving our customers well?

We want to keep our feedback loops short and tight, to keep signal-to-noise ratio high. It means that people building the systems must also operate them. If you own a part of the codebase, you also own the service it provides. This means being responsible for functional and non-functional characteristics of a feature. Ownership applies from feature creation to deprecation. To enable that, we need a shared toolkit for troubleshooting, monitoring and alerting.

Infrastructure monitoring and some operational metrics run on Zabbix. We are adopting Graphite/Grafana to keep an eye on business operations aspects of the system.

We use New Relic to check the HTTP endpoints. It's not cheap but works well, thanks to its instrumentation capabilities. We've defined performance and error alerts for our higher volume endpoints. Respective teams get alerted via VictorOps rotations if performance degrades. New

Relic also provides some tracing features to see the time spent on different steps of request processing.

When an exception happens in production, it gets sent to Rollbar. Rollbar groups and counts the occurrences and sends alerts for spikes or new items. In general it allows us to spot glitches in the system and estimate how many customers they affect.

To analyze a specific problem or identify patterns in logs, we use the Elastic stack. The stack consists of LogStash, ElasticSearch and Kibana. They are, respectively, used to parse, index and search/visualize logs.

Slack helps us to channel alerts and comms into one place.

There's a video and slides covering some of this from Baltic DevOps 2016 where we were honored to speak.

Vincent has written a great post our way of doing post-mortems. We use them to learn from customer-affecting production issues.

Bits & pieces

Vahur wrapped up some of the other tooling that we use.

PHP - There's a few tools in the company built in PHP, some legacy, some purposefully kept in PHP.

Spark - While being a pretty buzzword-averse bunch of engineers, we do like to adopt the right tech for the right task. In our case the prime problem domain to benefit from machine learning has been our financial crime prevention subsystem. We're building up machine learning models on Apache Spark.

Ansible & Terraform - Our infrastructure is growing and humans make mistakes. That makes infrastructure a prime automation target. We're adopting Terraform for declarative infrastructure management. We use Ansible to configure the instances.

Context

We build TransferWise for our current and future customers. They don't care much about which libraries, frameworks and components we bring in to move their money. It's our job as engineers to pick the tools and practices that allow us to deliver value quickly and sustainably.

Often we've chosen the simplest tool to getting the job done, instead of the most powerful. In other times we've gone with a framework that's not the newest kid on the block but has wide production adoption.

We're all proper geeks. We love new tech. We love to play and learn, pilot and break stuff. We do so in our hobby projects and geek-out sessions over Coke and beer. When building for our customers, we optimize for their happiness over the chance of adopting the next left-pad 0.0.3 ;) And you're right - it's as boring as building the HyperLoop of financial networks for millions of people could ever be.