SOFTWAREMILL

A reactive platform for real-time asset trading

Case Study

CLIENT PROFILE

Our client is a marketplace for buying and selling many currencies as well as ICO (Initial Coin Offering).

PROJECT OVERVIEW

INDUSTRY
FINTECH

GOAL OF THE PROJECT
IMPLEMENTING CORE ENGINE FOR A DIGITAL ASSET EXCHANGE SYSTEM

CHALLENGE
HANDLE UP TO 10000 REQUESTS PER SECOND IN A TRADING PLATFORM

PROJECT DURATION
2.5 YEARS, INCLUDING 10 MONTHS TO REWRITE THE TRADING SERVICE

TECHNOLOGIES
SCALA, AKKA PERSISTENCE, AKKA CLUSTER, AKKA STREAMS, CASSANDRA (DSE), APACHE KAFKA, GATLING, PROMETHEUS, GRAYLOG, DC/OS

Scala akka cassandra kafka Gatling

BENEFITS

- REACTIVE AND RELIABLE EXCHANGE SYSTEM WITH INCREASING VOLUME
- EVENT STORE KEEPING ALL BUSINESS EVENTS, ENABLING REAL-TIME INSIGHTS AND REPORTING
- MODULARITY ALLOWING KEEPING THE EXCHANGE BUSINESS RUNNING WHILE UPDATING OTHER COMPONENTS

BACKGROUND

Our customers' system was a very successful one, mainly because it has been developed in a very short period of time, providing key features like limit offers, market offers, extended history search, and other important capabilities. Exposed as an elegant and robust web interface as well in a form of accessible REST API, the platform quickly became the main source of transactions for traders all over the world.

As the user base started to expand, the company has been frequently adding new attractive markets. This, combined with solid marketing oriented on trustworthiness and reliability, has driven the company to an extraordinary success.

However, it has become more and more expensive for the company to handle failures manually. Additionally, a very high growth of users and markets started to show that performance capabilities are limited. Because of very simple design, the system was very well doing its job, but only to a point when it became necessary to scale it.

The main reason was that it was based only on the Akka Actor model with a few actors and their mutable state backed by some blocking calls to the infrastructure layer. It was impossible to put it in a multi-node environment without a heavy rework, requiring deep knowledge about the internals.

Additionally, test coverage was very basic, which left no room for confidence in case of very invasive updates. It was also clear that the system requires a strong division between handling commands - offer submissions, and queries - requests for secondary, derivative data (read model).

CHALLENGES

Primary drivers for starting a new project was to resolve current scalability problems in such a way, that adding new markets and handling increasing traffic becomes a matter of spinning up new nodes in a cluster. Online trading systems attract both regular users who put their offers using a web interface, as well as advanced traders, who extensively use the direct API with bots. In case of exceptional events, markets can quickly become flooded with a heavy load of requests, and the system is expected to process them quickly and correctly. Therefore, absolutely top priority was to design the core trading engine for data consistency.

Order books, and events resulting from offer matching needed to be the absolutely reliable source of truth, consistent with our users' wallets. Lags, lost messages, downtimes and similar problems may quickly reduce the trustworthiness of a platform, especially that such issues often cause immediate money losses.

The nature of trading platforms makes them notably attractive to hackers and scammers, making security another crucial component. That's why we've put so much effort into testing on all possible levels.

However, some parts of the system have been identified as secondary, especially reports and some of the views presented to users. This means that we could require only eventual consistency in those parts of the system, and clearly separate them using CQRS principles (Command/Query responsibility segregation).

After successful deployment, our team continued to add new features and new microservices, benefitting from established architecture and communication patterns.

SOLUTION

Our engineers started with thorough study of the existing system. In order to build a new one with the same API, we began to implement end-to-end tests, which have been configured to run against both old and the new solution.

In parallel, we took all the performance and reliability requirements and proposed an event-based architecture. Main exchange core was based on Akka Cluster and Akka Persistence. With clustering, we could shard the business logic into markets, running on separate nodes and allowing great scalability.

With clear segregation between commands and queries, we were able to write separate services for processing the core business logic of offer matching, and the definitions of so-called projections: pipelines for event processing and building the read model in various storage types.

Many types of projections were initially left to be just the same as in the original system, just now we were writing to these databases using projectors. Such separate projector services are Kafka consumers, so we can easily manage parallelisation and separation which particular projections run on which nodes. This way we separated expensive and crucial projections (like transfers, or building the main view of orderbooks) from secondary ones - like history search views.

Another set of projector nodes have been configured separately only to handle incoming queries. This way redeployments of projections or main engine didn't affect requests for data.

Our DevOps engineers prepared a solid build pipeline with infrastructure defined as a code for all the environments like development, staging, and production. In the meantime, we connected Prometheus and Grafana for monitoring, as well as Graylog for log aggregation. Then we designed detailed performance tests using Gatling, to see how our setup could handle 10000 requests per second.

SOLUTION

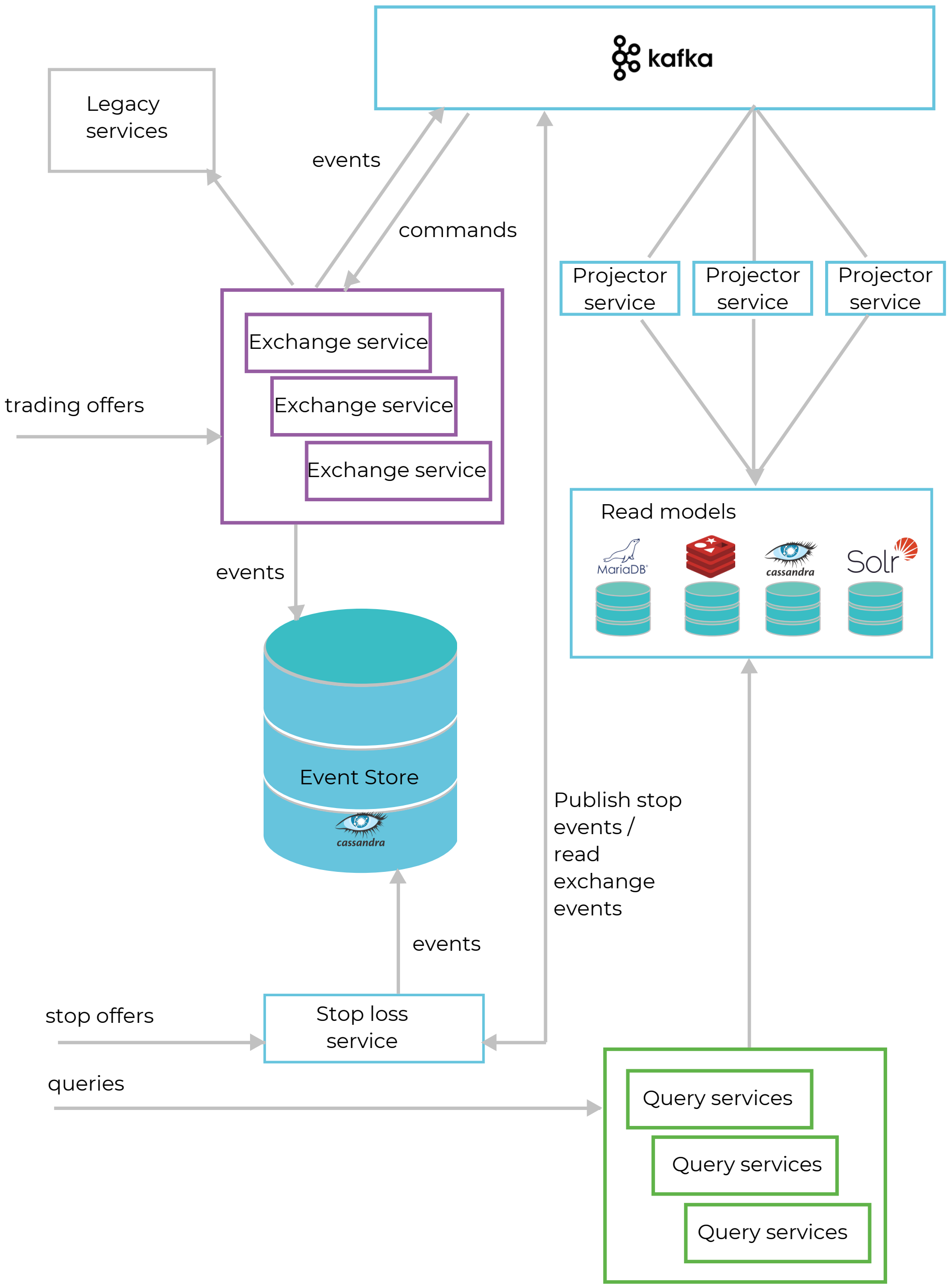
When the initial version was ready, we could collect metrics from Grafana and Gatling itself to identify potential bottlenecks and see what are the platform capabilities. It turned out that we could easily handle thousands of messages with a few nodes, and adding more nodes to the Akka Cluster would increase the throughput without problems. Some existing legacy services in the customer's system turned out to be slowing down the processing to hundreds of messages per second, but this was a satisfying result for a start. We could move on to rewrite these services in later phases.

Finally, we defined a DSL which allowed us to write many acceptance tests for offer matching scenarios. With a large set of such tests passing for both old and new platforms, we were getting closer to the release. Last step required QA engineers to test all the remaining scenarios manually, and to do some safe tests on production, at least for the read-only part.After successful release the new platform turned out to be robust and correct, and we could start developing new features, like handling of Stop Offers.

This new module was capable of reading events from the exchange core and submitting commands to using Kafka. Thanks to an advanced monitoring and alerting setup, we could also offer an attractive support plan for our customer.

With such a plan, they could be sure that any failures are quickly addressed by the on call team. The team was, at the other hand, very confident that many kinds of failures would be automatically recovered due to reactive mechanisms like restarts, at-least-once-delivery guarantees, replication, and isolation.

SOLUTION



kafka
Legacy services
events
commands
Projector service
Projector service
Projector service
Exchange service
Exchange service
Exchange service
trading offers
Read models
MariaDB
cassandra
Solr
events
Event Store
cassandra
Publish stop events / read exchange events
events
stop offers
Stop loss service
queries
Query services
Query services
Query services

RESULTS

The exchange platform not only continued to generate growing revenue to our customer, but gave them strong confidence that the system is ready for unexpected events and that it can process large volumes.

Also, updating the platform with fixes and new features has become a safe, frequent action affecting only minimal scope, and keeping the rest up and running, so that trading could continue even during redeployments.

We started to design new services, which have been implemented and deployed with equal success. Our engineers also worked on rewriting some of the legacy services in order to make them more reactive and match the architectural idea.

The solution gives our customer a possibility to quickly react and as many new markets as they want, without worrying about technical capabilities.

New API users can join in great numbers, including large companies who connect to the API with high-frequency automatic trading algorithms.

SOFTWAREMILL

GOT AN IDEA?
WE'LL MAKE IT HAPPEN

contact@softwaremill.com
www.softwaremill.com

We are SoftwareMill, a Poland(EU)-based consulting & custom software development company, delivering services remotely, worldwide for 10 years. Being experts in Scala (Akka, Play, Spark), Java, Kotlin we specialize in blockchain, distributed, big data systems, machine learning, IoT, and data analytics.

We believe that focus on quality, self-improvement and a true engineering approach can result in systems that do their job, bring value to clients, help them scale and grow.