From Monolithic to Microservices

An Experience Report from the Banking Domain

Source: https://www.computer.org/csdl/mags/so/2018/03/mso2018030050.html

Antonio Bucchiarone
14-18 minutes

Nicola Dragoni
Schahram Dustdar
Stephan T. Larsen
Manuel Mazzara

Abstract—This experience report of a real-world case study from the banking domain demonstrates how reimplementing a monolithic architecture into microservices improves scalability. The case study is based on Danske Bank's FX Core currency conversion system.

Keywords—microservices; software architecture; scalability; software development; software engineering; Danske Bank; FX Core

MICROSERVICES[1–3] ARE AN architectural style that originated from service-oriented architecture (SOA),[4] with the idea of bringing into the small (within an application) those concepts that worked in the large—i.e., for cross-organization business-to-business workflow. The shift toward microservices is a sensitive matter these days, seeing as how several companies are involved in a major refactoring of their back-end systems to accommodate the advantages of the new paradigm. This is the case for the system and the institution considered in this article—i.e., the FX Core of Danske Bank. (FX stands for foreign exchange, which is also called *forex*. It is the exchange of currencies—i.e., the conversion from one currency to another.)

In monolithic architectures, the modularization abstractions rely on the sharing of resources of the same machine (memory, databases, or files), and the components are therefore not independently executable. A notable problem of monoliths involves scalability and, in general, all the aspects related to change.[5] In the microservice paradigm, a system is structured by composing small independent building blocks, each with a dedicated persistence tool and communicating exclusively via message passing. In this kind of organization, the complexity is moved to the level of coordination of services.

Each microservice is expected to implement a single business capability—in fact, a very limited system functionality—bringing benefits in terms of service scalability. Since each microservice represents a single business capability, which is delivered and updated independently, discovering bugs or adding minor improvements does not have any impact on other services and on their releases. In common practice, it is also expected that a single

service can be developed and managed by a single team.[2,6] The idea to have a team working on a single microservice is rather appealing: to build a system with a modular and loosely coupled design, you should pay attention to the organization structure and its communication patterns because they, according to Conway's law,[7] directly impact the produced design. So, if you create an organization with each team working on a single service, that structure will make the communication more efficient not only on the team level but also within the whole organization, improving the resulting design in terms of modularity.

Microservices is not just another name for SOA. Indeed, there are some notable differences. In SOA, services are not required to be self-contained, with data, a user interface, and their own persistence tools—e.g., a database.

SOA does not focus on independent deployment units and related consequences; it is simply an approach for business-to-business intercommunication. The idea of SOA was to enable business-level programming through business process engines and languages such as WS-BPEL (Web Services Business Process Execution Language) and BPMN (Business Process Model and Notation) that were built on top of the vast literature on business modeling.[8] Furthermore, the emphasis was all on service orchestration rather than on service development and deployment.

In this article, we report the experience of migration from monolithic to microservices of Danske Bank's FX Core system. The documentation of the original system architecture was sparse, and the vast majority of technical details were obtained by direct conversations and interviews with the FX Core team and by manually inspecting the source code.

Migration Process

The migration process was business-driven and outside-in; i.e., **the system was designed and implemented one business functionality at a time.** The business functionalities were defined mostly by communicating with stakeholders (FX traders) and were iteratively added according to the level of priority for the business itself. We considered case by case whether a functionality should result in a new service or not. **If the business functionality seemed isolated and big enough, or it was shared among numerous other business functionalities, then it resulted in a new service.** In some cases, some functionalities might have been initially included together in a service and only later moved into their own separate services. For example, this occurred when the functionality was too big or was equally required by multiple services.

One of the benefits of this approach is that it has distanced the team from the old implementation, hindering the possibility of reimplementing everything as a distributed monolith.

Danske Bank's FX Core System

FX encompasses everything from private transactions performed in foreign countries (e.g., Internet shopping from abroad and the use of credit cards while traveling) to corporations moving their financial assets from one currency to another and exporting or importing products to and from foreign markets.

FX has grown with globalization and is now the largest financial market in the world, averaging a daily transaction volume of roughly five trillion dollars. This results in some transactions reaching hundreds of millions of dollars. Unlike the stock exchange, there is no centralized market. Instead, FX is decentralized and done over the counter; i.e., traders negotiate prices and trade directly between each other. Traders are typically the largest multinational banks, trading on behalf of their customers or themselves. Additionally, due to the decentralized and global nature of FX, the market is open 24 hours a day, five days a week.[9]

The FX IT system (see Figure 1) is part of the bank's Corporates and Institutions (C&I) department, and it acts as a gateway between the international markets and Danske Bank's clients and traders. C&I's clients are mainly large financial institutions and large multinational corporations. The FX Core system is part of FX IT; it handles trades and line checks—i.e., checking whether a client has the financial collateral (e.g., stocks, bonds, or cash) to perform a trade and how a trade will affect that collateral. This includes registration, validation, and post-trade management.

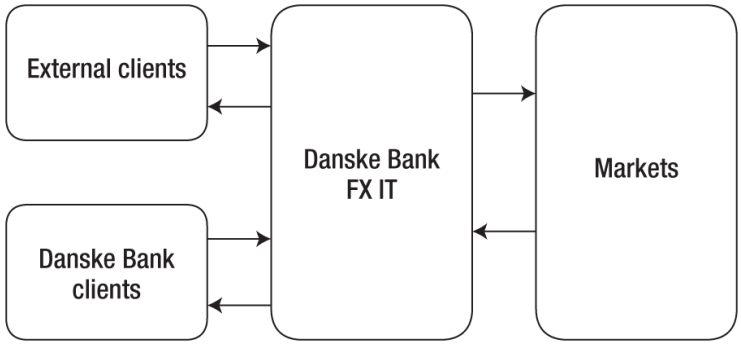

Figure 1. The FX IT system. FX stands for foreign exchange.

So that the information can be publicly available, all the confidential details such as concrete names of protocols, external providers, and specific services have been withheld. Furthermore, the internal logic of certain components cannot be described in depth.

FX Core Microservice Architecture

Danske Bank's new FX Core architecture is based on the microservice architectural style and is intended to completely replace the old monolithic architecture (see Figure 2).

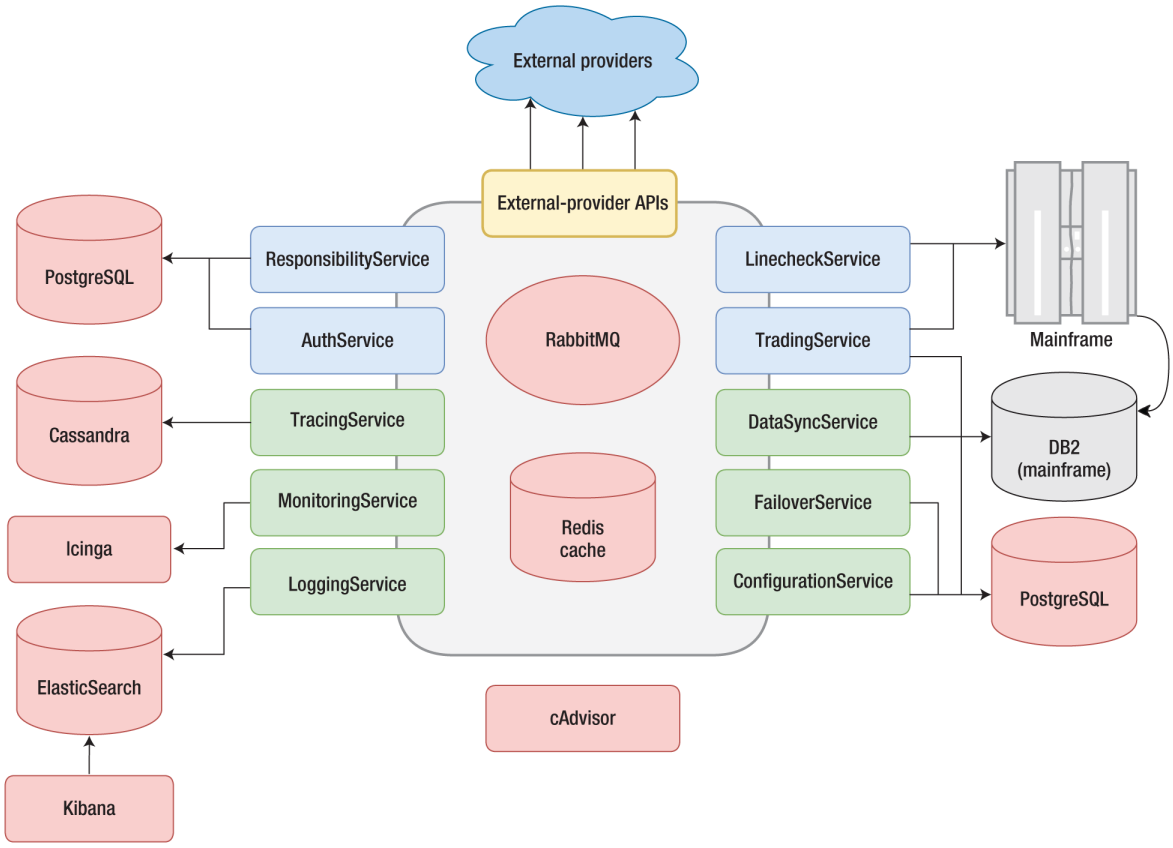

Figure 2. The new FX Core microservice architecture.

The experience of migration, explained next, is an example of how this kind of architecture can be implemented in an enterprise setting. Here, we identify major improvements of the migration, regarding both architectural aspects and company processes:

- *Containerization/Docker*. All services are now hosted in Linux containers on the Docker Swarm Cluster. This enables the use of the suite of tools provided by Docker. For example, the system uses Docker Compose to deploy the whole architecture with a single command. All these operations had to be done manually before. Also, **all container images are hosted in an internal Docker Registry, a central repository**. New images are deployed to the internal registry when a new version of a service is successfully built and tested by the continuous-integration system.
- *Automation*. All the services in the architecture now have an automated continuous integration and continuous deployment (CICD) pipeline. The tooling coming with Docker Swarm has an API enabling automation of several infrastructural tasks—e.g, rolling updates—that can be utilized within the CICD pipeline. Modern DevOps approaches for CI and CD can now be applied, bringing full automation to the process and following the DevOps corporate philosophy of reducing barriers between teams. Dockers allows supporting such an approach.
- *Orchestration*. Docker Swarm allows orchestration. **Failed services can be automatically restarted, providing self-healing. The orchestration tooling also allows service discovery and load balancing.**
- *Integration*. **Services now integrate via message-passing choreography, with RabbitMQ as the messaging system** (see Figure 2).

On the IT department's roadmap for the internal datacenters is the adoption of the Red Hat OpenShift (www.openshift.com) Iaas/PaaS (infrastructure as a service/platform as a service) platform. However, at the moment, the infrastructure consists of virtual machines that are ordered through a web portal and are set up manually by the FX Core team.

Solving Monolithic Problems

Let us now see how the microservice architecture has improved or solved some of the problems identified in the monolithic architecture.

The large components of the monolithic architecture, which **were highly coupled, had overlapping responsibilities, and were integrated in a multitude of ways**, have been substituted with several independent microservices. Just the names of the services reveal their responsibility, and they are generally much smaller compared to the large monolithic services. They do not integrate directly, resulting in looser coupling and less chance of feature overlapping in the future. For example, in the monolithic architecture, trade registration and line checks were handled by both ForexAPI and RequestService. In the microservice architecture, TradingService and LineCheckService are handling these tasks individually, instead.

The monolithic architecture had many shared components, but in the microservice architecture, this has been reduced to only one shared component—the Lambda framework. Lambda is very minimal and is only meant to be a framework to connect to the infrastructure and provide standard formatting methods for, e.g., messages, logs, and health checks.

Due to the criticality of the information involved and the clearance necessary to take action, the mainframe will still be attached to the microservice architecture for some time to come. But, over time, the functionalities from the mainframe will be implemented as new services. This will in the future result in all FX functionality being extracted, totally decoupling the mainframe from the system. For now, the impact of the mainframe has been reduced by caching.

Since the microservices are independent, loosely coupled, and isolated components, they can be deployed individually, without affecting the other components. This makes deployment very simple, and the usage of Docker and Linux containers ensures that services run in the same environment during local testing, on test servers, and in production.

The whole reimplementation allows the team to kill all paths into the system that they do not control. Since the team controls the whole infrastructure with Docker, including databases and ports open to outside clients, the team can eliminate all unwanted access. This allows the team to develop open APIs for clients and traders in the bank to use, thus eliminating direct database queries and the like. This gives the team full ownership and control of internal implementation details.

Internally, the microservices integrate only via messaging on RabbitMQ. Due to using message-based choreography, the services do not call each other directly, thus resulting in very low coupling and no interfaces to violate. The system does communicate to external systems via other paradigms, such as proprietary protocols to external providers and, in the future, REST APIs. (REST stands for Representational State Transfer.)

The team aimed for a polyglot architecture, meaning that it is not technology dependent. The team is no longer dependent on the .NET platform or MS SQL databases and can implement the services in whatever language it likes. You might argue that the team is just becoming dependent on other technologies, such as Docker, but Linux containers are becoming a standard through the Open Container Initiative (www.opencontainers.org).

The microservice architecture has centralized logging in the form of LoggingService, ElasticSearch, and Kibana, allowing for the aggregation of logs from all services. The same applies to monitoring implemented with MonitoringService, Icinga, and cAdvisor, allowing for aggregated monitoring of metrics. Centralizing and aggregating both logs and monitoring gives the team a complete system status overview, allowing it to act proactively on suspicious and faulty behavior.

We have all learned that introducing this amount of distribution to a system results in a whole new range of problems to solve. Scalability,[10] loose coupling, and high coherence (and other microservice benefits) were almost given from the get-go, as we can simply replicate services to scale, and they are split up nicely according to domain boundaries. Now, aspects like fault-tolerance mechanisms, concurrency handling, and monitoring are of increasing importance. This situation has been valuable, as these aspects, when solved, create value in the system. With the monolith, these problems never occurred; the problems there were mostly about solving object-oriented complexity and deploying such a monster. We have also experienced how important infrastructure and automation is, since there are so many moving parts we need to both manage and connect.

The future will see growing attention regarding the matters discussed here and the development of new programming languages intended to address the microservice paradigm.[11] Languages for microservices should be able to model microservices in a uniform way and at a level of abstraction that also allows for their easy interconnection.[12] In a system like the one described in this article, the remodeling of part of the system would be significantly simpler if the programming language used natively offers microservice as a first-class entity (just think about the existence of large, highly coupled components).

Microservice composition techniques are needed; they have to be used when

- frequent revision of microservices is needed,
- changes occur in existing offered functionalities (i.e., microservice behavior), and
- adjustment of business policies and objectives (i.e., composition requirements) is required.[13]

This is a typical situation in a dynamic market such as FX, with continuously changing policies.