

Using Microservices for Legacy Software Modernization

Holger Knoche and Wilhelm Hasselbring, Kiel University

// Microservices promise high maintainability, making them an interesting option for software modernization. This article presents a migration process to decompose an application into microservices, and presents experiences from applying this process in a legacy modernization project. //



MICROSERVICES¹ HAVE GAINED attention as an architectural style for highly scalable applications running in the cloud. However, our discussions with practitioners show that microservices are also seen as a promising architectural style for applications for which scalability is not (yet) a priority. For business applications, which are typically in

service for many years, maintainability is of particular importance. With respect to maintainability, microservices promise an improvement over traditional monoliths due to their smaller code bases, strong component isolation, and organization around business capabilities. Furthermore, the team autonomy fostered by microservices is likely to


reduce coordination effort and improve team productivity.

It is therefore not surprising that companies are considering microservice adoption as a viable option for modernizing their existing software assets. Although some companies have succeeded in a complete rewrite of their applications,² incremental approaches are commonly preferred that gradually decompose the existing application into microservices.³ Other approaches to modernization—e.g., restructuring and refactoring of existing legacy applications—are also valid options.⁴ However, decomposing a large, complex application is far from trivial. Even seemingly easy questions like “Where should I start?” or “What services do I need?” can actually be very hard to answer.

In this article, we present a process to modernize a large existing software application using microservice principles, and report on experiences from implementing it in an ongoing industrial modernization project. We particularly focus on the process of actually decomposing the existing application, and point out best practices as well as challenges and pitfalls that practitioners should watch out for.

The Legacy System

The legacy system to be modernized is the customer management application of an insurance company, which provides customer data such as names and postal addresses to the specific insurance applications. It was initially developed in the 1970s and 1980s as a terminal-based application on a mainframe using the Cobol programming language. In the early 2000s, user interfaces based on Java Swing were added, with the underlying logic being implemented



in Cobol on Unix servers. In recent years, new applications have been developed solely in Java, and a service infrastructure has been introduced to access the functionality on the mainframe directly from Java.

The customer application is currently used by a total of 35 client applications, ranging across the entire technology stack. These applications access the customer application in different ways, some by using defined interfaces, but many by invoking internal modules or even accessing the underlying database tables directly.

In addition to the technological complexity, the scope of the application has widened over the years. New functionality not native to the customer domain has been added, leading to further growth and complexity. As of today, the customer application consists of about one million lines of code in 1,200 Cobol modules.

Modernization Drivers and Goals

The primary driver for modernizing the customer application is the fact that it has become increasingly difficult to deliver new features on time. Since a large, high-priority project requires fundamental changes to large parts of the application, this lack of evolvability is considered a strategic risk. According to the developers, there are two main reasons for this low evolvability—namely, a deterioration of the application's internal structure and the high number of entry points for client applications. This has made the impact of changes difficult to assess, leading to a high amount of testing and rework. Secondary modernization drivers are the vendor and technology lock-in as well as the fact that many developers are close to retirement and Cobol developers are difficult to obtain.

The modernization goals are therefore as follows:

- establishing well-defined, platform-independent interfaces based on the bounded contexts of the underlying domain;
- improving the evolvability of the application—in particular, reducing the number of entry points and preventing access to internals, moving noncustomer functionality into separate components, and eliminating redundant and obsolete parts of the application; and
- an incremental platform migration from Cobol to Java.

Microservices are a suitable architecture for achieving these goals due to their organization around business capabilities, high evolvability, strong component separation, and focus on cross-platform interaction.

In order to choose a viable modernization process, several options were assessed with respect to their financial, technical, and staff-related feasibility. One of the fundamental questions of this assessment was whether to immediately implement the new services on the Java-based technology stack or to first build Cobol implementations and gradually replace them after migrating the client applications. As the platform migration from Cobol to Java was considered a high technical risk, the latter option was chosen to separate the client migration from the platform migration.

The Modernization Process

The proposed modernization process consists of five steps and is shown schematically in Figure 1. For the sake of conciseness, we show

only two client applications, A and B, both of which invoke the customer application's functionality by directly calling modules. In addition, module B₁ from B directly accesses one of the customer database tables as part of a query—for instance, a join. This initial situation is depicted in Figure 1a. The remaining steps are described next, together with experiences from implementing them in the aforementioned modernization project.

Step 1: Defining an External Service Facade

The first step of the modernization process is concerned with defining an external service facade that captures the functionality required by the client systems in the form of well-defined service operations (see Figure 1b). The implementation of these operations is performed in later steps.

The major challenge of this step is to define domain-oriented services that provide the functionality needed by the clients, but without conserving questionable design decisions from the legacy application. We approached this challenge from two sides. First, we created a target domain model and used it to define service operations from scratch that we expected to be provided by the application. Afterward, we employed static analysis to identify the “entry points” of the application—i.e., programs, methods, or database tables that were accessed from other applications. This was indispensable, as nobody was aware of all the accesses. Once we knew the entry points, we proceeded to analyze the invoked functionality and to formulate it as provisional service operations.

Then, similar or redundant operations were merged, and the

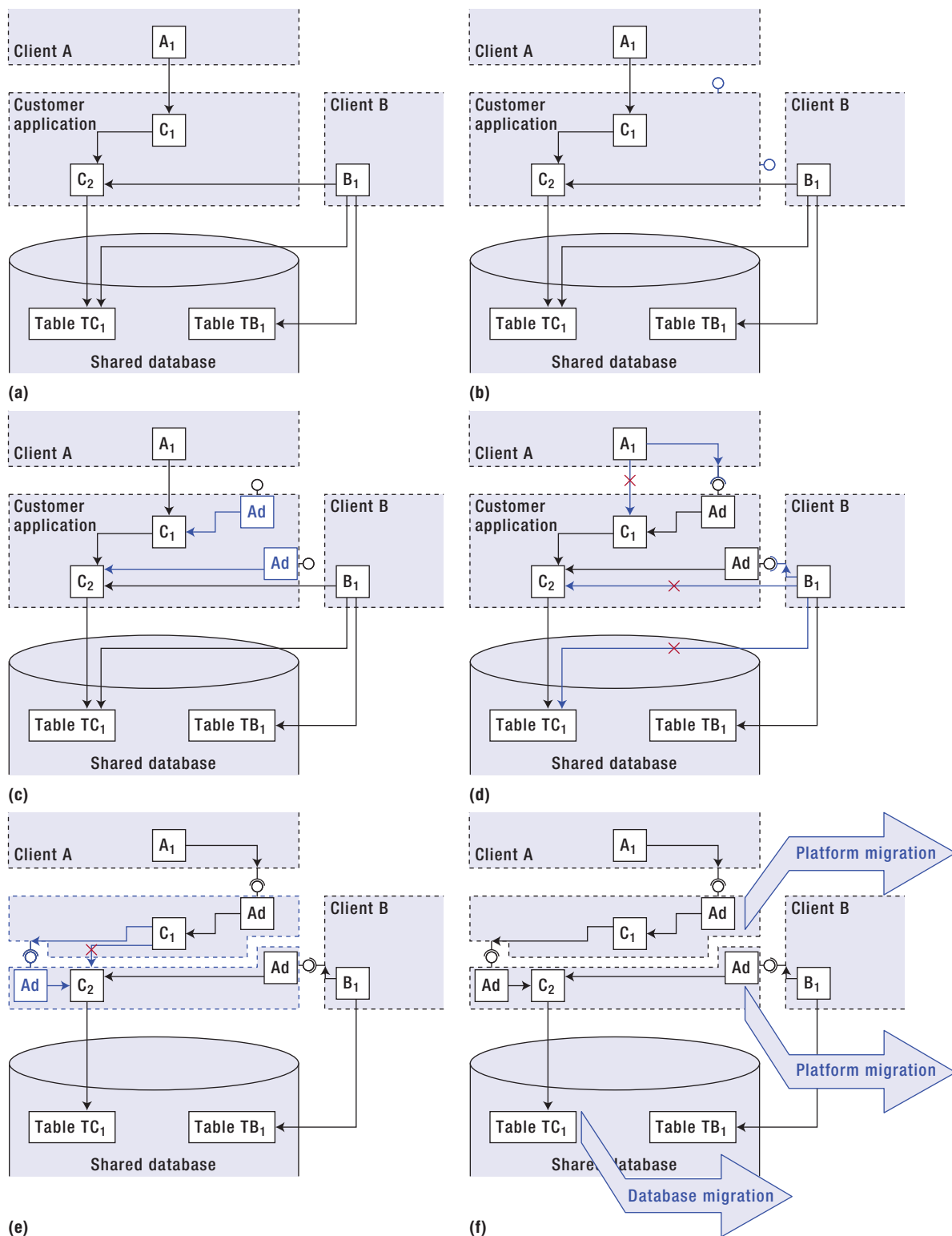


FIGURE 1. Overview of the modernization process. (a) The initial situation. (b) Defining an external service facade. (c) Adapting the service facade. (d) Migrating clients to the service facade. (e) Establishing internal service facades. (f) Replacing the service implementations by microservices. Changes in the respective process steps are highlighted in blue.

provisional operations were semantically matched against the expected operations. Ultimately, 29 service interfaces with about 150 service operations were derived from more than 500 entry points.

An important advantage of this approach is that in addition to the actual facade, it also produces information on how to replace the existing entry points with service operations, and provides candidates for adaptation. This information is used extensively in subsequent steps.

Step 2: Adapting the Service Facade

After defining the service operations, implementations must be supplied. While it is possible to immediately provide microservice implementations, we chose to first build an implementation by adapting the existing system, as shown in Figure 1c. Thus, the client migration to the service facade and the platform migration, which both posed considerable risk, were split into separate steps, at the cost of creating a throwaway implementation.

A key challenge of this step is to find appropriate candidates for adaptation. For this task, the results from the entry point analysis from step 1 proved to be particularly helpful. However, due to the refinement of the service operations, several operations had to be implemented from scratch.

For a successful adaptation, it is imperative to ensure sufficient testing. This can be difficult in legacy environments, as such environments may provide little to no facilities for common testing techniques like mocking. The new microservices are usually much easier to test, as technologies like Docker Compose can be used to set up environments in an ad hoc fashion. In our modernization project, the lack of

mocking for the legacy database proved to be a particular challenge. All tests were therefore designed so that modifications to the test data were either rolled back or explicitly compensated.

Step 3: Migrating Clients to the Service Facade

Once the service operations are implemented, client applications can start migrating to the new facade by replacing their existing accesses with service invocations (see Figure 1d). This step poses organizational as well as technical challenges and usually consumes a large part of the overall project time and budget, since large parts of the client applications must be changed and tested.

In order to support the development teams during the migration, we created a transition documentation. This documentation contained a textual description of how to replace each of the entry points identified in step 1 with one or more service operations. For each of the new operations, detailed descriptions and code snippets were provided to facilitate the transition as much as possible. This documentation was considered very helpful by the developers.

For the actual migration, many client applications successfully employed client-side adapters that emulated specific parts of the old interfaces using the new service facade. Thus, the changes to the existing modules could be reduced significantly. Following the idea of the Tolerant Reader pattern,⁵ these adapters relied on only those fields and operations actually required for the respective application, thus preventing interface changes from trickling into the client applications. The idea of creating a shared adaptation facility for all client applications

was, however, discarded, as this would have preserved the highly complex interface structure the modernization was aiming to improve.

During the migration, a few service methods proved to be too finely grained, leading to performance degradation due to invocation overhead. Therefore, these methods had to be refined during the migration.

In our project, the client migration took almost two years. To keep track of the overall migration progress, we employed the static-analysis toolset used in step 1, regularly creating a report of modules still using the old entry points. This report was then compared to the migration plans of the client applications to ensure that the migration proceeded as planned.

Step 4: Establishing Internal Service Facades

The techniques described in steps 1 to 3 can also be used to restructure applications internally (see Figure 1e). Although this restructuring can be done in parallel with establishing the external service facade, we decided to perform this step separately, as performing both restructurings at the same time was considered both too risky and resource-demanding. Since no naming conventions could be exploited, all programs were inspected manually and assigned to the appropriate components. Programs containing functionality from different components were assigned to all the respective components and marked for separation. Potentially obsolete programs were flagged for later deletion.

Step 5: Replacing the Service Implementations with Microservices

Once all desired service facades are established, the process of actually introducing microservices can begin,

FURTHER READING

Additional information on migrating to microservices can be found in numerous sources. Patterns and transition strategies are discussed in *Building Microservices*⁸ and *Migrating to Cloud-Native Application Architectures*.³ Detailed information on common pitfalls and antipatterns can be found in *Microservices AntiPatterns and Pitfalls*.⁹ Further industrial case studies are presented in “On Monoliths and Microservices”¹⁰ and “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture.”¹¹ Information for implementing microservices on specific platforms can be found, for instance, in *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*¹² and *Spring Microservices in Action*.¹³

as the adapted service implementations can now be transparently replaced (see Figure 1f). It is, however, important to note that several modernization goals have already been reached. Although the implementation is still based on the old Cobol code, it is now only accessed using well-defined, platform-independent interfaces, and the application has been internally restructured into the desired components. In particular, the database has been disentangled so that, for instance, schema changes can now be performed without affecting client applications.

Although the platform migration is transparent in theory, there are numerous practical challenges. Two exemplary challenges are described next—namely, transactions and resilience.

Transactions are ubiquitous in business software, and many applications rely on the fact that all changes are automatically rolled back in case of failure. For distributed architectures such as microservices, transactions are notoriously difficult to implement, and transactionless approaches such as explicit compensation are preferred.

However, when microservice implementations are introduced that do not provide the same transactional guarantees as the former implementation, the potential inconsistencies must be investigated, and compensation needs to be added to the client application if necessary.

Distributed architectures are furthermore susceptible to partial failure, a problem that does not arise in monolithic applications. Therefore, many applications rely on the availability of their dependencies. In such situations, the ability to cope with unavailable dependencies needs to be established before the replacement—for instance, by inserting circuit breakers.⁶

Current Situation, Further Steps, and Limitations

As of now, our modernization project has been running for almost four years. The client migration is almost completed and is running successfully in production. Furthermore, the first new service operations have been implemented, and the first batch of requirements for the strategic project was delivered on time.

In total, new implementations were created for 23 service operations, several of which, though, still have to access the Cobol code to remain compatible with operations that have not yet been migrated. These implementations are not yet true microservices, as several technological and organizational barriers still have to be overcome. In particular, the issue of transactional guarantees is still subject to discussion, which is why mostly read-only operations have been migrated so far. We observe, however, that the first careful steps toward infrastructure automation and DevOps practices⁷ are being taken in the wake of the migration, as the new implementations create opportunities for experimenting with these approaches.

As for the time spent on the individual process steps, the definition and adaptation of the service facade took 10 and 15 months, respectively. The client migration took about 20 months, and the creation of the new service implementations has taken about nine months.

It should, however, also be noted that certain parts of the application cannot be modernized using the presented approach. In particular, some user interfaces, which are built on highly proprietary technologies, lack the necessary means for service abstraction. As these user interfaces are embedded by other client applications, they must be preserved for the time being.

In this article, we have presented a process for decomposing an existing software asset into microservices based on our experience from an industrial case study. Obviously, this article can only scratch the surface of this complex

matter. Therefore, additional material on the topic has been compiled in the “Further Reading” sidebar.

Even though our modernization is not yet complete, we have already reached essential goals, as we were able to eliminate uncontrolled access to the internals of the application and we met the first batch of new requirements on time. Furthermore, practices like automated testing and code reviews have been established as a by-product of the modernization, which is considered a notable success.

In retrospect, we conclude that despite the additional cost of creating throwaway implementations, separating the **client migration from the platform migration was the right thing to do in this particular project**, as several technical challenges are still not solved. For projects with a less risky platform migration, however, this effort is not justified, and new service implementations should be immediately provided as microservices. Furthermore, due to the effort involved, this process is viable only for migrating large, complex software systems with a high business value. 🍷

References

1. J. Lewis and M. Fowler, “Microservices,” 2014; martinowler.com/articles/microservices.html.
2. W. Hasselbring and G. Steinacker, “Microservice Architectures for Scalability, Agility and Reliability in E-Commerce,” *Proc. IEEE Int’l Conf. Software Architecture Workshops* (ICSAW 17), 2017, pp. 243–246.
3. M. Stine, *Migrating to Cloud-Native Application Architectures*, O’Reilly, 2015.
4. W. Hasselbring et al., “The Dublo Architecture Pattern for Smooth Migration of Business Information Systems,” *Proc. 26th Int’l Conf. Software Eng. (ICSE 04)*, 2004, pp. 117–126.
5. M. Fowler, “Tolerant Reader,” 9 May 2011; martinowler.com/bliki/TolerantReader.html.
6. M. Nygard, *Release It! Design and Deploy Production-Ready Software*, Pragmatic Bookshelf, 2007.
7. L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect’s Perspective*, Addison-Wesley, 2015.
8. S. Newman, *Building Microservices*, O’Reilly, 2015.
9. M. Richards, *Microservices AntiPatterns and Pitfalls*, O’Reilly, 2015.
10. G. Steinacker, “On Monoliths and Microservices,” 2015; dev.otto.de/2015/09/30/on-monoliths-and-microservices.
11. A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture,” *IEEE Software*, vol. 33, no. 3, 2016, pp. 42–52.
12. B. Familiar, *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*, Apress, 2015.
13. J. Carnell, *Spring Microservices in Action*, Manning, 2017.

ABOUT THE AUTHORS



HOLGER KNOCHE is a senior software architect at b+m Informatik and a PhD student in Kiel University’s Software Engineering Group. His research interests include software architecture and software modernization, especially runtime performance and data consistency during the move toward decentralized architectures such as microservices. Knoche received a master’s in computer science from FHDW Hannover. He’s a member of the German Association for Computer Science. Contact him at hkn@informatik.uni-kiel.de.



WILHELM HASSELBRING is a professor of software engineering at Kiel University. In the Software Systems Engineering competence cluster, he coordinates technology-transfer projects with industry. His research interests include software engineering and distributed systems, particularly software architecture design and evaluation. Hasselbring received a PhD in computer science from the University of Dortmund. He’s a member of ACM, the IEEE Computer Society, and the German Association for Computer Science. Contact him at hasselbring@email.uni-kiel.de.

myCS

Read your subscriptions through the myCS publications portal at

<http://mycs.computer.org>