16 Mar, 2017

Using Microservices for Legacy System Modernization

Share:

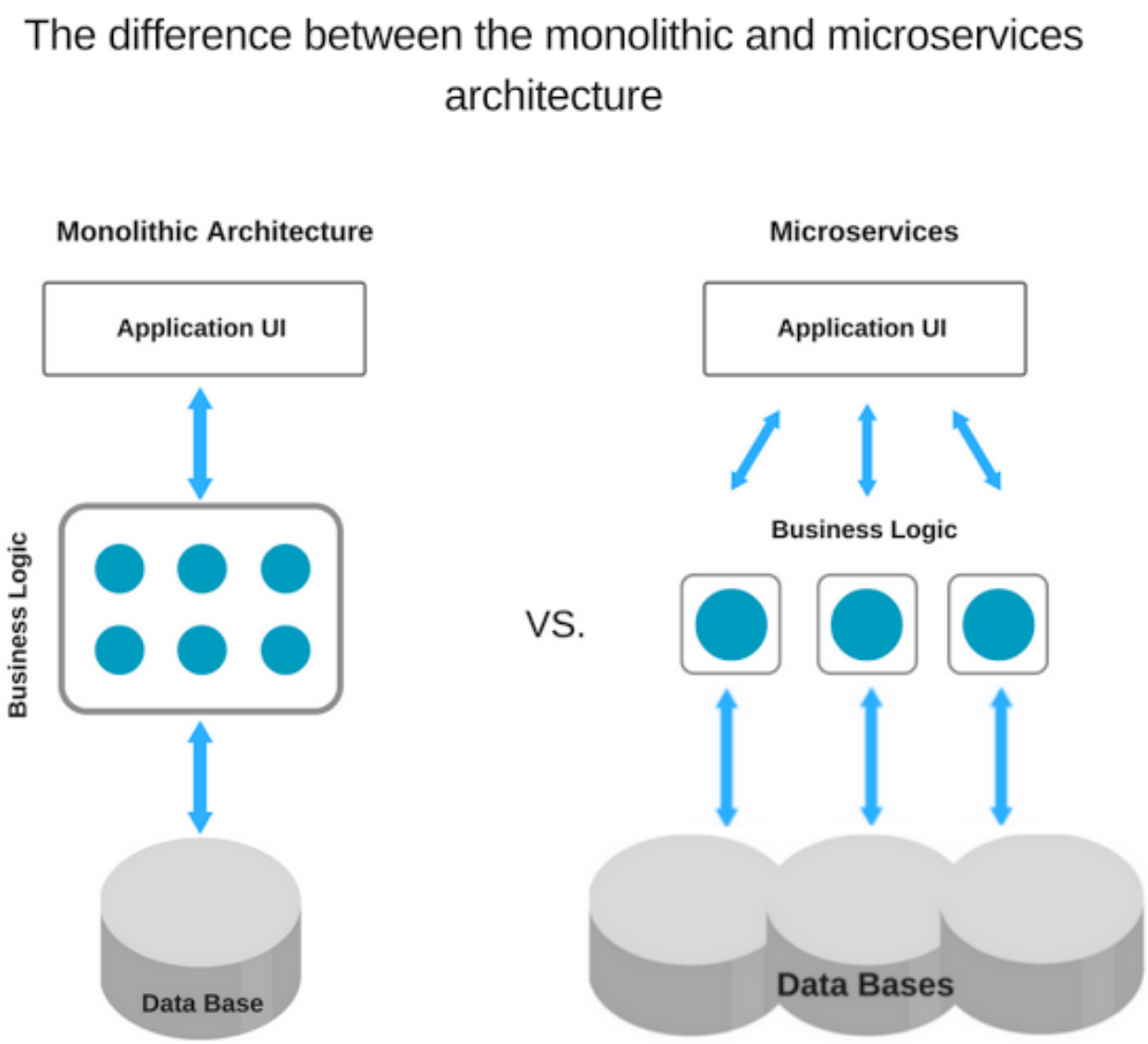
From 1974 to 1996, German professor Meir Lehman and his Hungarian colleague László Bélády had been formulating eight laws of the software evolution. Their statements seem obvious to us now. For example, the Law of Declining Quality says that the quality of software will decline unless it is maintained and adapted to environment changes. Nevertheless, now their observations serve as an instruction that helps us define a legacy system and understand when it's time to modernize.

Some techniques for software evolution are more common than others: automated migration, commercial, off-the-shelf software, re-hosting, code refactoring, and architecture review (e.g. here's how AltexSoft applied this approach), and, of course, the topic of today's discussion–microservices.

What's the Difference between Monolithic and Microservices Architecture?

To understand microservices, let's first discuss the concept of monolithic architecture. The monolith is a software development pattern where all code components are interconnected and interdependent. Be it user interface or backend services, all elements are compiled together. Thus, developers cannot work separately and need to coordinate efforts so they don't disrupt each other's projects.

That's how most legacy software is built. It doesn't necessarily mean the monolith is bad. It has its own benefits, such as easier engineering and testing, but it has a barrier to adopting new technologies. Also, the monolithic codebase can be too large to make changes fast and correctly, making it not the best approach to modernization.



In contrast, microservices is a method of software application development in which the system is composed of small independent services, each running as a separate process. Here, services can be considered as features or modules that your product consists of. These can be an account service, a product catalog, or an email subscription manager. In terms of technology, microservices are nothing new, but they provide a completely different approach to physically separate the branches of business logic. Such giants as Netflix, eBay, Amazon and Twitter have been using microservices to handle high loads and implement complex systems for years.

Why Use Microservices for Legacy System Modernization?

If you've been noticing the signs that it's time to modernize your software and looked for the ways to do it, microservices are among the most popular approaches. There are a number of factors why they've gained such popularity. Let's have a look at them.

- **Small, autonomous teams allow for better communication**
  Microservices by definition are independent elements and must be manipulated as such. Thus, they are usually developed and managed by small teams of up to eight people (often called two-

pizza groups). This organizational structure not only helps developers become attuned to their codebases and resolve issues faster, but also provides better in-group communication.

- **Independent deployment doesn't require synchronization of processes**
  When you deploy each service independently, outages in a single segment of an application will less likely bring down an entire application. Consequently, even if some services are down, most of the clients won't notice, and the team can resolve problems without rushing. Also, during the deployment process, developers don't need to coordinate local changes which enables continuous deployment, saves time, and resources.

- **Elements can be scaled separately**
  Having monolithic architecture, you are required to scale all components together, compromising on the choice of hardware. Microservices allow you to deal only with functional bottlenecks, scale the parts that have performance issues, and use the hardware that best matches service requirements. Moreover, the automated scaling allows for normalizing configuration during off-peak hours, which results in a better customer experience and cost-savings.

- **Each microservice is developed using the most fitting technology**
  Developers can choose whatever language or technology is best suited for each service. Plus, when the services are small, it's easier to rewrite them using new and modern technologies. With such continuous modernization, your system won't become quickly outdated. "*The nice thing about microservices from a company's standpoint is that they allow for higher developer satisfaction and higher developer retention. Because, due to the fact that microservices are self-encapsulated, it is possible to give developers more freedom of choice which frameworks, libraries, programming languages, tools, etc. they want to work with.*" says Renat Zubairov, CEO of elastic.io, the microservices-based hybrid integration platform.

- **Phased implementation helps escape complete rewriting**
  With microservice architecture, you can isolate a small part from your existing application and replace it with a microservice. By breaking a monolith into pieces, you don't have to completely reimplement it, but rather identify one or more clearly defined functional chunks and pull them out as microservices. However, this approach can be successfully used only at the later stages of migration and will produce a number of complications for developers at the beginning of microservices implementation. We'll be discussing that in the next section.

Possible Hurdles to Consider

Nothing comes without a price. Many believe that although microservices are great in theory, in practice there some difficulties with legacy software modernization. Here are the drawbacks to consider.

- **The engineering effort may set you back**
  One of the main challenges of refactoring to microservices is to gradually replace functionality while minimizing the changes that must be added to support this transition. Since microservices imply a distributed system, developers need to think of the ways to extract the function from the interconnected monolith and keep the new services linked to the remainder of the system. Not only does this increase the amount of work for developers, but it also may slow down the whole development process.

- **Inter-service communication creates more errors**
  As opposed to the monolithic architecture, a microservices-based application is a distributed system that incorporates multiple modules. Each microservice uses some sort of an API to exchange data. Changes to the API can introduce errors, such as message format differences between API versions. Also, as more components attempt to exchange information, network congestion can occur.

- **Testing and monitoring may be daunting**
  With monolithic architecture, we can test the entire codebase. With microservices, each independent service needs to be tested separately. It's a complex and protracted process. However, the problem can be eliminated with a set of modern approaches, such as contract testing or the end-to-end method.

- **You will have to adopt DevOps**
  If your company doesn't embrace the DevOps practice, you will lose the main reason to go with microservices–delivering scalable applications fast. To ensure this, development and operations must be aligned to work jointly on a problem, so that one would be able to keep pace with the another. However, considering the popularity of DevOps, this problem can be transformed into an opportunity. Then, you can take on not one, but two evolutionary approaches that perfectly complement each other.

Three Strategies of Microservices Implementation

Fortunately, developers have been refactoring monolithic systems long enough to establish a number of successful practices. Each of the following strategies is based on the Strangler Application approach, a term coined by a software developer Martin Fowler.

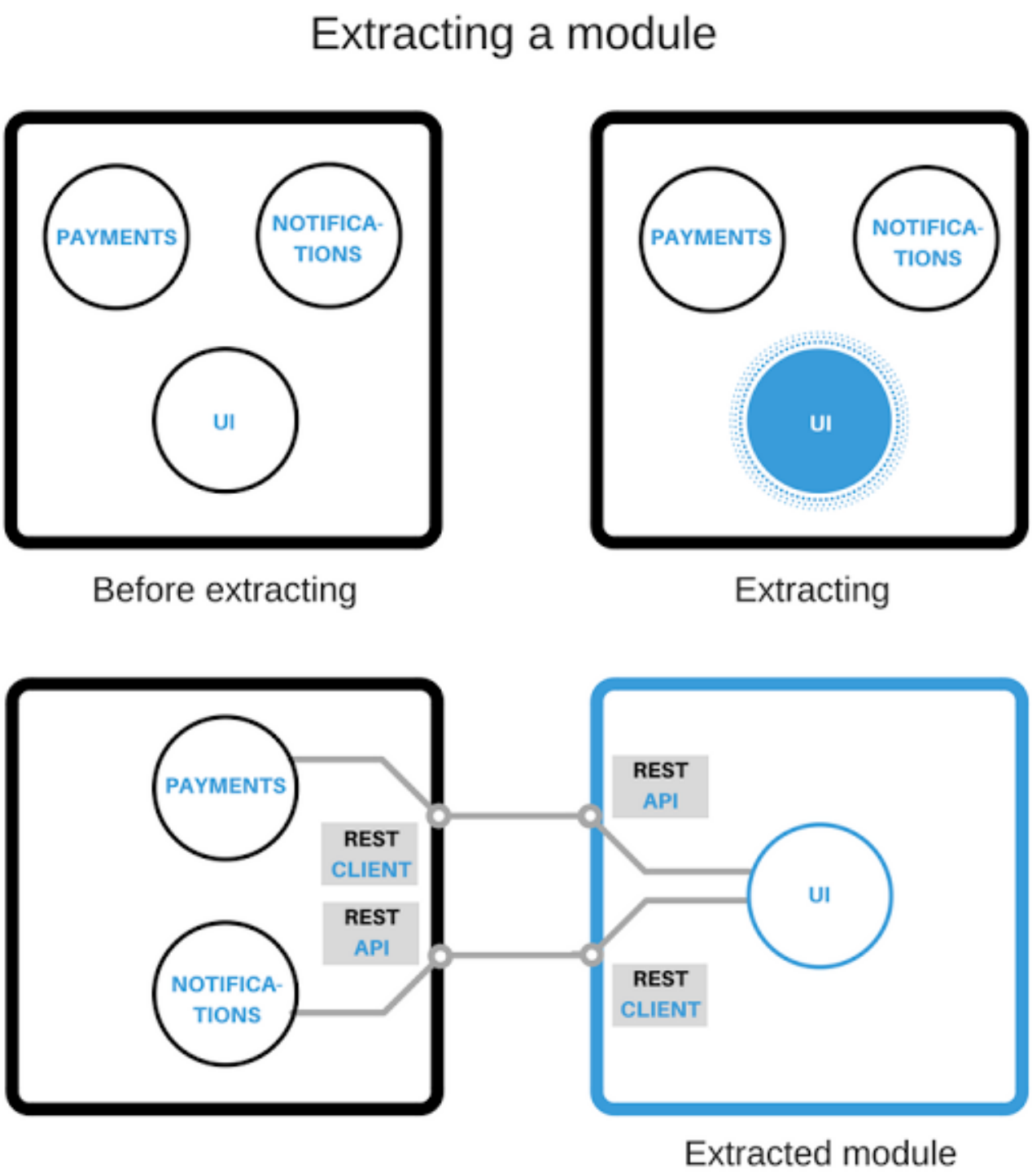
Strangler vine, found in rainforests, is known for growing around a tree, and when the tree dies, it leaves a tree-shaped vine behind. These strategies imply using the same pattern. By building a new application based on microservices and running it simultaneously with the monolithic one, you shrink your legacy application so that it eventually becomes another microservice.

- **Implement each new functionality as a microservice**
  When implementing a new functionality, don't add it to your existing monolith. Instead, put it in a

microservice. This approach prevents a monolith from becoming bigger and harder to deal with, allowing each new service experience the benefits of the microservice architecture. However, this strategy doesn't help to eliminate a monolith, and the following approaches address that.

- **Break down the monolith**
  Usually, a monolith application consists of three components: user interface (front-end), business logic (back-end), and databases. The natural separation between them allows for splitting a monolith into smaller applications, thus making it easy to develop and scale applications independently. Still, it's possible that two applications (front-end and back-end + databases) will be just as unmanageable as one, and this technique can be used only as a step towards a complete containerization.

- **Extract modules into microservices**
  The most logical strategy to use is to extract already existing modules within a monolith into standalone microservices. Once you've extracted enough modules, the monolith will stop being a monolith. The following diagram illustrates the refactoring process.



Here's a quick note: don't focus on just one of these strategies. You can also find a way to combine and implement them together to create your own hybrid strategy.

Microservices Architecture Use Cases

Before microservices became such a big topic of discussion, some companies adopted the technology that hadn't been named yet. Let's see how it worked out.

- **Netflix**
  Being one of the earliest adopters of the microservices architecture, Netflix shared their journey before the term even existed. In 2008, the company was growing rapidly and was unable to build

data centers to keep up with scale. Plus, the smallest error in the code required all Netflix engineers to engage into finding a problem. After refactoring non-customer facing applications to microservices in 2009, Netflix began working with customer-facing services, such as account signup, movie and TV selections, and others. By the end of 2011, they'd fully and successfully broken up their monolith into hundreds of microservices. Today, its website is powered by 500+ microservices and 30+ independent engineering teams.

- **Walmart**
  Having to deal with 6 million pageviews per minute, Walmart had to rearchitect their giant online business from monolith in 2012. Their huge step to digital transformation required changing the organization's approach to DevOps and migrating the system to microservices. As a result, the company increased their mobile orders by 98 percent and raised conversions by 20 percent overnight. The biggest concern–downtimes during peak events such as Black Friday–was also eliminated, along with no downtime altogether after refactoring.

- **Amazon**
  In 2001, Amazon.com was an architectural monolith. Although hundreds of developers were working on a very small piece of the website, they still had to coordinate each change with other members' projects. Each update or bugfix had to be synchronized. The company even establish a "merge Friday"– the day when all developers merged their changes and created a final version that would go into production. This discouraging process added a lot of frustration, so the company made an important decision and pulled out functional units to wrap them with a web service interface. Moreover, their two-pizza teams were given full ownership of one or a few microservices, which increased responsibility and efficiency. Now, thanks to continuous delivery, Amazon is able to make 50 million deployments per year.

- **Spotify**
  With the intention to stay innovative in a highly competitive market, Spotify required an architecture that could scale independently to 75 million users. Microservices allowed them just that, plus solved another problem–the lack of constant synchronization within the company. By creating autonomous, full-stack teams, each consisting of front- and back-end developers, testers and UI designers, Spotify increased responsibility for each operation and got rid of overlapping between different tasks. The company was also able to create a seamless experience for millions of users by having many services down without customers even noticing it.

Conclusion

Of course, you don't have to follow the path just because it's been trod upon by hundreds of others. But when the world is turning its gaze in one direction, it's worth taking a look.

Microservices provide some truly advantageous possibilities and in view of the large community behind the technology, the drawbacks are gradually disappearing. Other migration techniques,

regardless of their benefits, fail to produce the same combination of speed, scalability, and in-team communication while staying agile in the quickly changing world.

Legacy system evolution is a long journey, but with microservices you can undertake the task piece by piece, while taking a step in the direction of digital transformation.

Further Reading

13 Signs Your Legacy Systems Need Modernization

AltexSoft Helps Merlot Aero Advance Airline Management by Enhancing the Legacy System and Building up New Features for its Transportation SaaS Product