Journey from Monolith Application to Microservices

Ms. Nithya Rajagopalan
Director of Engineering, Strategic Procurement, SAP Ariba, Bangalore, India
Nithya.Rajagopalan@sap.com

Mr. Kumaraswamy Gowda
Principal Software Engineer, SAP Ariba, Bangalore, India
kumaraswamy.m@gmail.com

Traditional monolithic architecture

Any software application has three main components – an user interface, Data access layer and Datastore/database. A monolithic system is one large system that has all the three tightly coupled and deployed together. In a monolithic application, the application logic, user interface, backend tasks/jobs are all in one huge code base. While in certain circumstances monolithic applications are preferred, but there are many problems with them.

Problems with monolithic application

- Tight coupling between the different layers
- Resilience: If one part of the system fails, it could bring down entire system
- Scale: Scale everything even if a particular component needs improvement
- Deployment: Even if there is one-line of change, deploy entire system
- Complex development team structure
- Technology: Adopting newer technologies like programming languages, databases, framework

All the above problems make the software application inflexible for expansion, unscalable for complex applications and blocks continuous development.

What are microservices

At a very high level, microservices can be seen as a way to create independent applications, where applications are broken down into smaller, independent services based on domain and functionality. Each microservice is not dependent upon a specific programming language and hence allows different services to be developed with different technologies that gives best results. This makes each microservice independent and self-contained offering a single functionality in a bigger scheme of the whole application.

Monolithic vs Microservices architecture

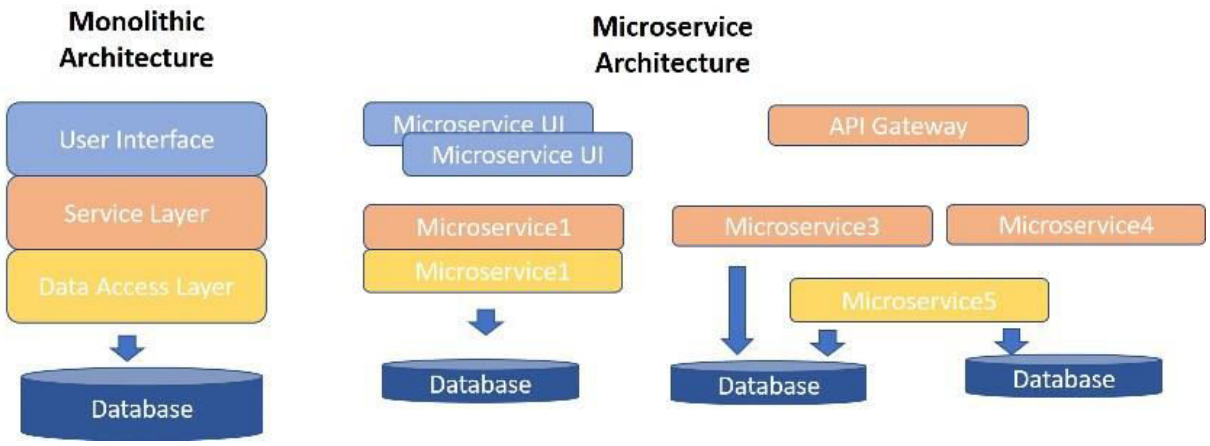

Figure 1Monolith vs Microservices architecture

Benefits with Microservices

Decoupling

Microservices enables us to make different functionalities to be loosely decoupled. Each microservice use their own execution environment but communicate to each other through REST APIs or a Message Bus. Service Discovery helps to search and find the route of communication.

Granular Scaling

Individual services can horizontally scale up or down in seconds and in an auto-scaling manner. Since each service is independent, they can be independently built, modified and scaled without having to scale the whole application. This indirectly improves performance of particular functionalities with minimum investment in resources.

Business continuity and Fault isolation

Microservice architecture is built for isolating failures and respond to outages without hampering the whole system. If there is a partial functionality or defect in one microservice, the rest of functionalities of the application can still continue, thus helping to meet SLA's for most customers.

Composability

One of the key promises of distributed systems and service-oriented architectures is to open up opportunities for reuse of functionality. With microservices, we allow our functionality to be composed and decomposed in different ways for different purposes. Each service can be aggregated using Aggregator pattern (with or without proxy) or chained using Chaining pattern(with or without branching).

Technology Heterogeneity

The composable ability of microservices also provide us flexibility with technology and team structures. It allows us to choose different technology for different services allowing to pick right tool for the job instead of having to select a more standardized one-size fit all approach. This enables faster adoption of newer technologies and autonomous development teams.

Patterns for moving from Monolith to Microservices

Domain Driven Design (DDD)

Before beginning any refactoring, figure out the domains of the application that helps to componentize in terms of business logic.

Large problem domains can be decomposed into sub-domains to manage complexity and to separate the important parts from the rest of the system.

- **Core Domains:** Core Domains must have a fundamental competitive advantage in the system and it should be the reason for the success of the system.
- **Generic Domain:** This is not the core, but the core depends on it. Examples are an e-mail sending service, notification service.

Bounded Contexts

A *bounded context* clarifies, encapsulates, and defines the specific responsibility to the model. It ensures the domain will not be distracted from the outside. Each model must have a context implicitly defined within a sub-domain, and every context defines boundaries.

The idea is that any given domain consists of multiple bounded contexts, and residing within each are things that do not need to be communicated outside as well as things that are shared externally with other bounded contexts. Each bounded context has an explicit interface, where it decides what models to share with other contexts.

Decouple components (Loose coupling)

When services are loosely coupled, a change to one service should not require a change to another. The whole point of a microservice is being able to make a change to one service and deploy it, without needing to change any other part of the system. This is really quite important.

- Create decoupled and independent components for each defined domain
- Each component serves a separate business requirement
- Rewrite new code and retire old code
- Analyze dependency between components
- HTTP REST & Service discovery for communication between microservices for minimal direct communication between services to ensure it has no direct reliability

Minimize dependency back to Monolith

A major benefit of microservices is to have a fast and independent release cycle. Having dependencies to the monolith - data, logic, APIs - couples the service to the monolith's release cycle, prohibiting this benefit.

One should establish some hard and fast rules for the kinds of dependencies that you are prepared to support. If you really cannot avoid referring back to a feature in the monolith do it through a service façade. This can provide an architectural placeholder for a future service implementation or at the very least act as an anti-corruption layer.

Strangler Pattern

The Strangler Pattern is a popular design pattern to incrementally transform your Monolith application into Microservices by replacing a particular functionality with a new service. Once the new functionality is ready – the old component is strangled, the new service is put into use & the old component is decommissioned all together.

You can develop a new component, let both the new and the old component exist for a period of time and finally terminate the old component once the new component is stable.

- Initially all application traffic is routed to the Legacy application.
- Once the new component is built, you can also test your new functionality in parallel by enable for small set of users against the existing monolithic code.
- Both the monolith and the new built component need to be functional for a period of time. Sometimes the transitional phase can last for an extended duration.
- When the new component has been stable, you can get rid of it from legacy monolithic application.

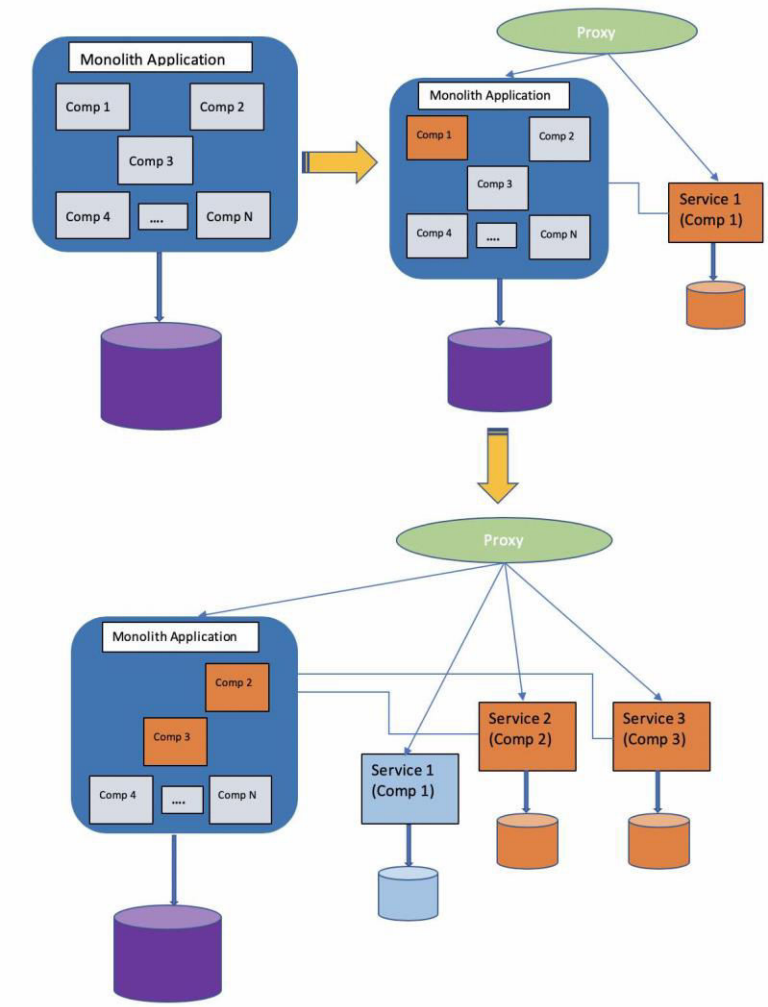

Figure 2: Strangler pattern

Feature Flag

With feature flags, engineering teams can have complete control over their various microservices. First, wrap the microservice with a feature flag, with all traffic going to the old version within the monolith application. Then, release the microservice with the feature flag ON, gradually put whatever traffic you want to the new microservice for only specific set of users, similar to a blue green deployment. Once the new microservice is stable, open up the solution for larger set of users However, feature flags can serve arbitrarily complex (or simple) variations of traffic to the new microservice.

API Gateway and Database per service

When a monolithic application is moved to microservice architecture, the database needs to be designed in such a way that it is forward looking. The best option is to keep the service self-sufficient and ensure it has its own data. This would mean that some parts of the data from the monolithic service needs to be moved. The advantage of this is that the service becomes a self-sufficient unit which can be accessed through

- API gateway - The application composes the whole by calling APIs from different services. Database transaction management can be done by Saga pattern.
- Materialized views using the CQRS pattern

Event publishing

Nearly every microservice needs to publish events when the data is updated. The Event Sourcing pattern provides a way to maintain a queue of events from the feeds that it receives. These feeds can be further aggregated in case a Domain Event model is suitable.

Challenges of Microservices

Monitoring

Breaking the system up into smaller, fine-grained microservices results in multiple benefits. It also adds complexity when it comes to monitoring the system in production. Monitoring an application based on a microservice architecture is different from monitoring a monolithic application

- Multiple servers to monitor with respect to CPU, Memory, health etc
- Multiple logs to sift through to point out where the error occurred
    - To grab our logs and make them available centrally
    - A solution could be to use Logstash or splunk along with Kibana elastic search for viewing logs. Kibana can even generate graphs from the logs you send it, allowing you to see at a glance how many errors have been generated over time
- Multiple places where network latency could cause problems
- Application metrics: These metrics relate specifically to your application. These top-level data are useful for development teams and the organization to understand the functional behavior of the system.
- Platform metrics

Monitoring requirements should be considered from the very beginning of an application's lifecycle. Systems monitoring requires contributions from both development and operations. It's a critical part of the operational support of any distributed system. Microservice architectures are even more distributed than a typical monolithic application. They require more real-time attention and proactive monitoring.

Authentication and Authorization

A common approach to authentication and authorization is to use some sort of single sign-on (SSO) solution. SAML and OpenID Connect both provide capabilities in this area.

Once Authenticated, the user principal will be provide information about all roles and access a user has. Some of the common solutions are:

Common Single Sign-On Implementations

- Identity provider could be an externally hosted system, or something inside your own organization.
- It is common to have your own identity provider, which may be linked to your company's directory service. A directory service could be something like the Lightweight Directory Access Protocol (LDAP) or Active Directory.
- SAML is a SOAP-based standard. OpenID Connect is a standard that has emerged as a specific implementation of OAuth 2.0.

Single Sign-On Gateway

- Each service could decide to handle the redirection to, and handshaking with, the identity provider.
- Gateway to act as a proxy, sitting between your services and the outside world
- The idea is that we can centralize the behavior for redirecting the user and perform the handshake in only one place.

Fine-Grained Authorization

In order to enforce resource protection of who can perform certain transactions or who can access and use specific data over an API channel, the API gateway solution should be complemented and extended with a fine-grained authorization solution. By extending the API Gateway with a dynamic policy based authorization solution, organizations will be able to enforce resource specific access control.

Service-to-Service Authentication and Authorization

Along with authenticating users, you might need to allow other services to interact with your API. While client applications can provide users with a web sign-in prompt to submit their credentials, you need another approach for secure service-to-service communication. Some of the standard methods are

- HTTP(S) Basic Authentication
- Use SAML or OpenID Connect
- Client Certificates
- HMAC Over HTTP
- API Keys

Do you really need microservices?

While microservices help with reducing the cost of application maintenance and time to production, there are cases when monolithic application is more suitable. Monoliths have fewer cross cutting concerns and less initial operational overhead. Since they are tightly coupled, they are easier to test and deploy and could also have advantage in performance. If your project is new and has lot of unknowns or if your project needs a faster time to market, then Monoliths are the way to go.

Prematurely decomposing a system into microservices can be costly, especially if you are new to the domain. In many ways, having an existing codebase you want to decompose into microservices is much easier than trying to go to microservices from the beginning.

All frameworks and patterns have their own pros and cons. We have tried to provide the necessary framework to help you apply the general principles and make a choice on what is best for your organization.

References

- Building Microservices by Sam Newman
- https://microservices.io/
- https://martinfowler.com/articles/microservices.html

About the authors

Nithya is currently the Director of Engineering for Strategic Procurement related products in SAP Ariba. She is a technical leader is in cloud computing and machine learning. She has experience in various domains like Telecom, IoT, Procurement and Retail which accounts for the 7 patents that she has in her name. Her favourite hobby is birding and wildlife photography which she likes to do with her family. https://www.linkedin.com/in/nithyasprofile

Kumar holds a MTech degree in Computer Science from NIT Surathkal, India. He is currently working as a Principal software engineer for Strategic Procurement related products in SAP Ariba. He has a overall of 13+ years of development experience from organizations like Oracle, JP Morgan, IBM Software labs. His expertise are in building enterprise and cloud application for domains like pharmaceutical, software tools and procurement. His interests are towards Microservices architecture and Machine Learning algorithms. He loves going on long rides on bike and conquering peaks during weekend treks. https://www.linkedin.com/in/kumaraswamym

---

Microservice Architecture | Microservices Tutorial for Beginners

This Edureka's Microservices video at https://www.youtube.com/watch?v=L4aDJtPYI8M gives you detail of Microservices Architecture and how it is different from Monolithic Architecture. You will understand the concepts using a UBER case study. In this video, you will learn the following: 1. Monolithic Architecture; 2. Challenges Of Monolithic Architecture; 3. Microservice Architecture; 4. Microservice Features; and 5. Compare architectures using UBER case-study

Microservices

Microservices are a software development technique—a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. In a microservices architecture, services are fine-grained and the protocols are lightweight. The benefit of decomposing an application into different smaller services is that it improves modularity. This makes the application easier to understand, develop, test, and become more resilient to architecture erosion. It parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently.It also allows the architecture of an individual service to emerge through continuous refactoring.Microservice-based architectures enable continuous delivery and deployment. http://bit.ly/2WCNFrB