Varun Khadayate

A016

B. Tech (CsBs) TY

Aim

Case study on Microservices and implementation of microservices in application

Case study on Microservices

Microservices architecture is an approach in which a single application is composed of many loosely coupled and independently deployable smaller services.

What are microservices?

Microservices (or microservices architecture) are a cloud native architectural approach in which a single application is composed of many loosely coupled and independently deployable smaller components, or services. These services typically

- have their own technology stack, inclusive of the database and data management model;
- communicate with one another over a combination of REST APIs, event streaming, and message brokers; and
- are organized by business capability, with the line separating services often referred to as a bounded context.

While much of the discussion about microservices has revolved around architectural definitions and characteristics, their value can be more commonly understood through simple business and organizational benefits:

- Code can be updated more easily - new features or functionality can be added without touching the entire application
- Teams can use different stacks and different programming languages for different components.
- Components can be scaled independently of one another, reducing the waste and cost associated with having to scale entire applications because a single feature might be facing too much load.

Microservices might also be understood by what they are not. The two comparisons drawn most frequently with microservices architecture are **monolithic architecture** and **service-oriented architecture (SOA)**.

The difference between microservices and monolithic architecture is that microservices compose a single application from many smaller, loosely coupled services as opposed to the monolithic approach of a large, tightly coupled application

The differences between microservices and SOA can be a bit less clear. While technical contrasts can be drawn between microservices and SOA, especially around the role of the **enterprise service bus (ESB)**, it's easier to consider the difference as one of scope. SOA was an enterprise-wide effort to standardize the way all web services in an organization talk to and integrate with each other, whereas microservices architecture is application-specific.

How microservices benefit the organization

Microservices are likely to be at least as popular with executives and project leaders as with developers. This is one of the more unusual characteristics of microservices because architectural enthusiasm is typically reserved for software development teams. The reason for this is that microservices better reflect the way many business leaders want to structure and run their teams and development processes.

Here are just a few of the enterprise benefits of microservices.

Independently deployable

Perhaps the single most important characteristic of microservices is that because the services are smaller and independently deployable, it no longer requires an act of Congress in order to change a line of code or add a new feature in application.

Microservices' loose coupling also builds a degree of fault isolation and better resilience into applications. And the small size of the services, combined with their clear boundaries and communication patterns, makes it easier for new team members to understand the code base and contribute to it quickly—a clear benefit in terms of both speed and employee morale.

Right tool for the job

In traditional n-tier architecture patterns, an application typically shares a common stack, with a large, relational database supporting the entire application. This approach has several obvious drawbacks—the most significant of which is that every component of an application must share a common stack, data model and database even if there is a clear, better tool for the job for certain elements. It makes for bad architecture, and it's frustrating for developers who are constantly aware that a better, more efficient way to build these components is available.

Precise scaling

With microservices, individual services can be individually deployed—but they can be individually scaled, as well. The resulting benefit is obvious: Done correctly, microservices require less infrastructure than monolithic applications because they enable precise scaling of only the components that require it, instead of the entire application in the case of monolithic applications.

There are challenges, too

Microservices' significant benefits come with significant challenges. Moving from monolith to microservices means a lot more management complexity - a lot more services, created by a lot more teams, deployed in a lot more places. Problems in one service can cause, or be caused by, problems in other services. Logging data (used for monitoring and problem resolution) is more voluminous and can be inconsistent across services. New versions can cause backward compatibility issues. Applications involve more network connections, which means more opportunities for latency and connectivity issues. A DevOps approach (as you'll read below) can address many of these issues, but DevOps adoption has challenges of its own.

Microservices and cloud services

Microservices are not necessarily exclusively relevant to cloud computing but there are a few important reasons why they so frequently go together—reasons that go beyond microservices being a popular architectural style for new applications and the cloud being a popular hosting destination for new applications.

Among the primary benefits of microservices architecture are the utilization and cost benefits associated with deploying and scaling components individually. While these benefits would still be present to some extent with on-premises infrastructure, the combination of small, independently scalable components coupled with on-demand, pay-per-use infrastructure is where real cost optimizations can be found.

Secondly, and perhaps more importantly, another primary benefit of microservices is that each individual component can adopt the stack best suited to its specific job. Stack proliferation can lead to serious complexity and overhead when you manage it yourself but consuming the supporting stack as cloud services can dramatically minimize management challenges. Put another way, while it’s not impossible to roll your own microservices infrastructure, it’s not advisable, especially when just starting out.

Challenges Faced

The enterprises also face several challenges while adopting and using microservices:

Complexity

A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.

Development and testing

Writing a small service that relies on other dependent services requires a different approach than a writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.

Lack of governance

The decentralized approach to building microservices has advantages, but it can also lead to problems. You may end up with so many different languages and frameworks that the application becomes hard to maintain. It may be useful to put some project-wide standards in place, without overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.

Network congestion and latency

The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns like queue-based load levelling.

Data integrity

With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.

Management

To be successful with microservices requires a mature DevOps culture. Correlated logging across services can be challenging. Typically, logging must correlate multiple service calls for a single user operation.

Versioning

Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.

Skill set

Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.

Implementation Of Microservices in Application

While just about any modern tool or language can be used in a microservices architecture, there are a handful of core tools that have become essential and borderline definitional to microservices:

Containers, Docker, and Kubernetes

One of the key elements of a microservice is that it's generally pretty small. (There is no arbitrary amount of code that determines whether something is or isn't a microservice, but "micro" is right there in the name.)

API gateways

Microservices often communicate via API, especially when first establishing state. While it's true that clients and services can communicate with one another directly, API gateways are often a useful intermediary layer, especially as the number of services in an application grows over time. An API gateway acts as a reverse proxy for clients by routing requests, fanning out requests across multiple services, and providing additional security and authentication.

Messaging and event streaming

While best practice might be to design stateless services, state nonetheless exists and services need to be aware of it. And while an API call is often an effective way of initially establishing state for a given service, it's not a particularly effective way of staying up to date. A constant polling, "are we there yet?" approach to keeping services current simply isn't practical.

Serverless

Serverless architectures take some of the core cloud and microservices patterns to their logical conclusion. In the case of serverless, the unit of execution is not just a small service, but a function, which can often be just a few lines of code. The line separating a serverless function from a microservice is a blurry one, but functions are commonly understood to be even smaller than a microservice.

Enterprise benefits of microservices are:

Independently deployable

Perhaps the single most important characteristic of microservices is that because the services are smaller and independently deployable, it no longer requires an act of Congress in order to change a line of code or add a new feature in application. Microservices promise organizations an antidote to the visceral frustrations associated with small changes taking huge amounts of time. It doesn't require a Ph.D. in computer science to see or understand the value of an approach that better facilitates speed and agility. But speed isn't the only value of designing services this way. A common emerging organizational model is to bring together cross-functional teams around a business problem, service, or product. The microservices model fits neatly with this trend because it enables an organization to create small, cross-functional teams around one service or a collection of services and have them operate in an agile fashion. Microservices' loose coupling also builds a degree of fault isolation and better resilience into applications. And the small size of the services, combined with their clear boundaries and communication patterns, makes it easier for new team members to understand the code base and contribute to it quickly—a clear benefit in terms of both speed and employee morale.

Right tool for the job

In traditional n-tier architecture patterns, an application typically shares a common stack, with a large, relational database supporting the entire application. This approach has several obvious drawbacks—the most significant of which is that every component of an application must share a common stack, data model and database even if there is a clear, better tool for the job for certain elements. It makes for bad architecture, and it's frustrating for developers who are constantly aware that a better, more efficient way to build these components is available. By contrast, in a microservices model, components are deployed independently and communicate over some combination of REST, event streaming and message brokers—so it's possible for the stack of every individual service to be optimized for that service. Technology changes all the time, and an application composed of multiple, smaller services is much easier and less expensive to evolve with more desirable technology as it becomes available.

Precise scaling

With microservices, individual services can be individually deployed—but they can be individually scaled, as well. The resulting benefit is obvious: Done correctly, microservices require less infrastructure than monolithic applications because they enable precise scaling of only the components that require it, instead of the entire application in the case of monolithic applications.