Principles of Microservices

10TH SEP 2017

Authored by: Arlin D'almeida, Principal Solution Architect

Abstract

There are multiple strategies to transform a system estate into an estate that is faster to deliver new features, easy to maintain, cost-efficient and robust. Of the many architectural styles for digital transformation, the architectural style that is getting a lot of attention in recent years is 'Microservices'. This style is not completely new; it is a type of distributed architecture that has evolved from component-based architecture and service orientated architecture. It can be viewed as a *specialisation* of the overall service orientated architecture paradigm.

'Microservices' is essentially independently deployable fine-grain components. It is owned by a small team who are responsible for the design, development, deployment and operations of the component. It builds on the service orientated architecture concept of loosely coupled business services that encapsulates functional, technical and implementation complexity to other services, and interact with the rest of the system estate by exposing a simple REST APIs. The paradigm shift from service-oriented architecture to Microservices is the change in focus from 'reusability' and 'integration through a service bus' to 'decoupling a complete business functionality' with emphasis on the 'complete ownership' of the Microservice by a small team.

Key Principles of Microservices

Principle 1: Fine Grain Components

What is the right size for a Microservice?

Before determining the size of the Microservices, let me explain what meant by Microservices. The term Microservices can be misleading. A more correct term should be 'Micro-Component Architecture'. It is the decomposition of a component into fine-grain components. It is a common misconception that the Microservice is the decomposition of the Web Service or API into fine-grain services; the 'Service' in Microservices is not the same as Web Service. So essentially it is fine-grain components that expose one or more APIs.

The size of the Microservice (or Micro-Component) is difficult to quantify. There are no rules on how big or small the Microservice should be. There is no standard unit to define the size of a Microservice. However, a widely accepted standard is to define the size of the Microservice by the size of the team designing, implementing and monitoring the same. The commonly accepted team size is 6 to 10 people, following the two pizza rule.

It is important to note that this provides only an indication of the team size and it should not be followed blindly. It is more important to understand the rationale of relating the size of Microservices to team size. The objective of having small teams, is to maintain focus and develop a sense of ownership of the component end to end which includes implementation and monitoring of the business function. There can be larger teams managing more complex business functions or smaller teams managing simpler business functions.

Principle 2: Decomposition with Cohesion

How do I decompose the system estate into Microservices?

Microservices must be decomposed by business functions and not by technical functions. This follows **Conway's Law** with states that organisations most often design systems which mirror their communication structure. The structure of business teams plays a large influence in drawing the boundaries of a business function and the same boundaries get replicated within the system estate. It is, therefore, necessary to align with the business boundaries while decomposing

components into Microservices to avoid introducing unnecessary interactions between business teams.

There are different approaches to decompose a system by business functions, the most common approaches are,

1. Service Decomposition by Business Capabilities
2. Service Decomposition by Domain
3. Service Decomposition by TMF Open Digital Architecture Functions

Service Decomposition by Business Capabilities. The concept of 'business capability' is from business architecture modelling. It is basically what a team does within the business to generate value. This could be sales, order capture, credit check, service support etc. This approach will have alignment with the business functions and well as alignment with interactions of a business function. A drawback of decomposing an application using business capabilities is that some classes are difficult to decompose and these classes will be common across multiple services. For example; an Order Class will be used in Order Capture, Order Orchestration, and Order Delivery etc. This issue is addressed by Service Decomposition by Domain.

Service Decomposition by Domain. The concept of 'domain' is from domain-driven design. It is taking a bottom-up approach to decompose the system estate based on domain objects or business entities. Examples of these are account, contact, order, order-line etc. These objects are further grouped as aggregates which are a cluster of domain objects. For example, a customer is a combination of account and party, and order aggregate is a combination of order and order line. An aggregate will have one of its constituent objects be the aggregate root. Any interaction should only reference the aggregate root to ensure the integrity of the aggregate as a whole. A unit of work or transactions should not cross aggregate boundaries.

One of the techniques to identify aggregates and the transactions is 'Event Storming'. It follows a *workshop based methodology* where *domain experts* help with defining 'Domain Events'. A *domain event* is anything that happens that is of interest to a domain expert. This is followed by determining triggers that cause domain events. It considers all sources of the triggers, e.g. UI triggers,

internal business processes, external systems etc. This is then followed by identifying data aggregates that accept triggers to accomplish events. These data aggregations help shape up a domain referred to as '*bounded context*'.

The *aggregate* is a self-contained state machine that focuses on the concept of a single domain and the *bounded context* represents a collection of associated aggregates with an explicit interface to the wider world.

Service Decomposition by TMF Open Digital Architecture Functions: TMF Open Digital Architecture (ODA) - Functional Architecture, provides a functional decomposition of the telecom estate. It is important to consider this for the digital transformation of the BSS/OSS and leverage these functional units as well. The TMF-ODA is used as a standard reference by major telecom operators and software vendors. Ensuring alignment with the functional decomposition of the ODA will ensure alignment with the rest of the industry especially software vendors who provide commercial off the shelf products.

The objective of all of the above approaches is to define the scope of business function such that it has a minimal dependency on other services and thus reducing the need to build integration points between functionality that is intrinsically cohesive.

Principle 3: Loosely Coupled Interfaces

What is loose coupling and why do we need it for Microservices?

Microservice Architecture reiterates the notion of hiding technical and implementation complexities within the Microservice. It reinforces the key concepts of Abstraction and Encapsulation.

1. **Abstraction** aims to hide the complexities of how the service has been implemented to the external world. It could be implemented using any language or underlying technology.
2. **Encapsulation** aims to hide the details of the data. This includes the database itself and its characteristic e.g. RDBMS or NoSQL etc. and internal data entities.

Both abstraction and encapsulation leads to two things,

1. Creation of a Microservice that this black box to the outside world
2. Need for well-defined APIs, such that the API is the only means through which the outside world can communicate with the Microservice and its data.

Abstraction and encapsulation enable the creation of two 'Worlds'. The internal world of the Microservice and the external world that the Microservice interacts with. The internal world has the freedom to manage its design and operations. It has the freedom and independence to choose how it should implement a business function. It can choose the technology, database and language best suited for its business function and not depend or be influenced or inhibited by the technology choices of other services. This ensures that the Microservice is extendable to support future requirements.

What constitutes a Microservice?

Initial versions of Microservices included a business logic layer with a separate database exposed through a set of APIs. The business function was autonomous with complete control over its runtime environment and database schema. A separate database per Microservice provided the freedom to choose the right database designs e.g. NoSQL, RDBMS depending on the business function. The business function itself could be implemented using the most appropriate technologies and frameworks. However, limiting the Microservice business functions and database exposed through APIs results in the creation of a Monolith UI that would consume the underlying APIs. This is not ideal and can potentially cause implementation and delivery issues. An approach gaining popularity is Micro Frontend, where the UI is also part of the Microservice. Thus a Microservice is a full-stack component. Therefore a web page will comprise of UI served through multiple Microservices. This ensures complete autonomy of the Microservices.

Principle 4: Simplified Integration Layer

What did we learn from SOA?

One of the criticisms of Service Oriented Architecture was its over-dependency on a centralised Enterprise Service Bus. Most SOA implementations resulted in

implementing complex business logic in the ESB. This resulted in several extendibility issues.

Microservices uses the concept borrowed from the internet of having smart endpoints and dumb pipes. The business logic and functions are moved from the integration layer to the components. This keeps the integration layer thin focusing on technical integration functions such as service registry, discovery, message routing, API composition, and protocol translation and security. This ensures extendibility and scalability.

Principle 5: Independently Deployable

What is unique about Microservices?

Microservices should be independently deployable with the choice of its deployment patterns. The common deployment patterns are

1. Multiple Service Instance per Host
2. Deployment on Host
    a. Deployment on VM
    b. Deployment on Container
3. Serverless Deployment

Multiple Service Instance per Host

In this pattern, one or more physical/virtual host is provisioned and multiple service instances are deployed. Each service instance runs on one or more hosts. A key benefit of this approach is its resource usage is efficient as multiple service instances share the server and operating system. Another advantage of this pattern is that it is faster to deploy a service instance.

A key disadvantage is that there is very little separation of the service instances unless each service instance is modelled as a separate process. A service instance can consume all CPU and memory of the host.

Deployment on Host

In this pattern, each service instance is deployed on its host. There are two specialisations of this pattern

4. Deployment on VM
5. Deployment on Container

Deployment on VM

In this pattern, each service is packaged as a virtual machine image. Each service instance is created as a VM and is launched using that VM image. The key benefit is that each service instance runs in isolation on its own VM. The amount of CPU and memory is fixed per VM and cannot use resources from other services. Another advantage of deploying the service as a VM is that it encapsulates service instance implementation details. Once a service has been packaged as a VM it can be viewed as a black box.

The key disadvantage is less efficient resource utilisation. Each service instance has the overhead of an entire VM. Another disadvantage of this approach is that deploying a new instance of a service is usually slow. VM images are slow to build due to its size.

Deployment on Container

In this pattern, each service instance is deployed on its container. A container consists of multiple processes running in a sandbox. The advantages of containers are similar to VMs, but it is much lighter. It isolates service instances from each other. Container images are very fast to build.

Serverless Deployment

In this pattern, the service instance is deployed to a Function as a Service like AWS Lambda. The key benefits are the cost since you only pay for what you use and not pay for any idle time. The key disadvantage is cold start, which is latency experienced after function is triggered.