WHITE PAPER

Microservices in Practice - Key Architectural Concepts of an MSA

By Kasun Indrasiri, Senior Director, WSO2

July 2019

Table of Contents

1. Monolithic Architecture

Enterprise software applications are designed to facilitate numerous business requirements. Hence, a given software application offers hundreds of business capabilities and all such capabilities are generally piled into a single monolithic application. Enterprise resource planning (ERP), customer relationship management (CRM), and other software systems are good examples - they're built as monoliths with several hundreds of business capabilities. The deployment, troubleshooting, scaling, and upgrading of such software applications is a nightmare.

Service-oriented architecture (SOA) was designed to overcome the problems resulting from monolithic applications by introducing the concept of a 'service'. Hence, with SOA, a software application is designed as a combination of services. The SOA concept doesn't limit service implementation to be a monolith but its most popular implementation, web services, promotes a software application to be implemented as a monolith, which comprises of coarse-grained web services that run on the same runtime. Similar to monolithic software applications, these services have a habit of growing over time by accumulating various capabilities. This growth soon turns those applications into monolithic globs, which are no different from conventional monolithic applications.

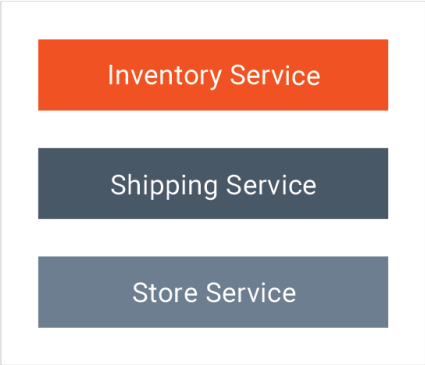

Figure 1: Monolithic Architecture

Figure 1 shows a retail software application that comprises of multiple services. All these services are deployed into the same monolithic runtime (such as application servers). Therefore, it shows several characteristics of a monolithic application: it's complex, and is designed, developed, and deployed as a single unit; it's hard to practice agile development and delivery methodologies; updating a part of the application requires redeployment of the entire thing.

There are a couple of other problems with this approach. A monolith has to be scaled as a single application and is difficult to scale with conflicting resource requirements (e.g. when one service requires more CPU, while the other requires more memory). One unstable service can bring the whole application down, and in general, it's hard to innovate and adopt new technologies and frameworks.

These characteristics are what led to the advent of microservice architecture. Let's examine how this works.

2. Microservice Architecture

The foundation of microservice architecture (MSA) is about developing a single application as a suite of small and independent services that are running in their own process, developed and deployed independently[1].

Most definitions of MSA explain it as an architectural concept focused on segregating the services available in the monolith into a set of independent services. However, microservices is not just about splitting the services available in a monolith into independent services.

Consider that by looking at the functionality offered from the monolith, we can identify the business capabilities required from the application - that is to say, what the application needs to do to be useful. Then those business capabilities can be implemented as fully independent, fine-grained, and self-contained (micro)services. They might be implemented on top of different technology stacks, but however done, each service would be addressing a very specific and limited business scope.

This way, the online retail system scenario that we introduced above can be realized with an MSA as depicted in Figure 2. As you can see, based on the business requirements, there is an additional microservice created from the original set of services that were there in the monolith. It's apparent, then, that this goes above merely splitting services onto more complex ground.

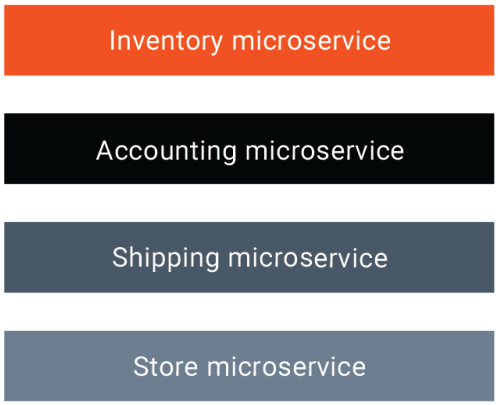

Figure 2: Microservice architecture

So let's examine the key architectural principles of microservices and, more importantly, let's focus on how they can be used in practice.

3. Designing Microservices: Size, Scope, and Capabilities

You're likely doing one of two things when it comes to microservices: you're either building your software application from scratch or you're converting an existing applications/services into microservices. Either way, it's important that you properly decide the size, scope and the capabilities of the microservices. This is perhaps the hardest thing that you initially encounter when you implement MSA in practice.

Here are some of the key practical concerns and misconceptions on the matter:

- Lines of code/team size are lousy metrics: There are several discussions on deciding the size of microservices based on the number of lines of code of the implementation or its team's size (i.e. two-pizza team). However, these are considered to be very impractical and lousy metrics, because we can still develop services that completely violate MSA principles with less code and two-pizza-teams.
- 'Micro' is a bit of a misleading term: Most developers tend to think that they should try to make the service as small as possible. This is a misinterpretation.
- In the SOA/web services context, services are often implemented at different granularities — from a few capabilities to several dozens of capabilities. Having web services and rebranding them as microservices is not going to give you any benefits of MSA.

Then how should we properly design services in an MSA?

3.1 Guidelines for Designing Microservices

- Single Responsibility Principle (SRP): Having a limited and focused business scope for a microservice helps us to meet the agility in development and delivery of services.
- During the designing phase, we should find the boundaries of the microservices and align them with the business capabilities (also known as bounded context in Domain-Driven-Design).
- Make sure the microservices design ensures the agile/independent development and deployment of the service. Your focus should be on the scope of the microservice, and not about making the service smaller.
- It is often a good practice to start with relatively broad service boundaries, to begin with, and then refactor to smaller ones (based on business requirements) as time goes on.

In our retail use case, you can find that we have split the capabilities of its monolith into four different microservices namely 'inventory', 'accounting', 'shipping' and 'store'. They are addressing a limited, but focused business scope, so that each service is fully decoupled from each other and ensures agility in development and deployment.

4. Messaging in Microservices

In monolithic applications, business capabilities of different processors/components are invoked using function calls or language-level method calls. In SOA, this was shifted towards a much more loosely coupled web service level messaging, which is primarily based on SOAP on top of different protocols, such as HTTP and JMS.

4.1 Synchronous Messaging - REST, gRPC, GraphQL, and Thrift

For synchronous messaging (the client expects a timely response from the service and waits to get it) in MSA, REpresentational State Transfer (REST) is the unanimous choice as it provides a simple messaging style implemented with HTTP request-response, based on resources. Therefore, most microservice implementations use HTTP along with resources

(every functionality is represented with a resource and operations carried out on top of those resources).

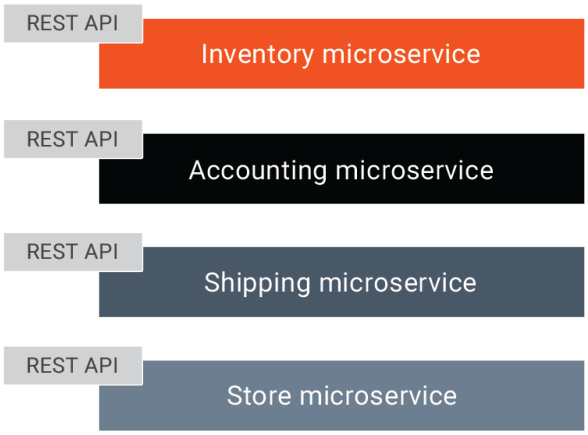

Figure 3: Using REST interfaces to expose microservices

gRPC is also getting quite popular as an inter-process communication technology and an alternative to RESTful services. It allows you to connect, invoke, operate and debug distributed heterogeneous applications as easily as making a local function call. Unlike REST, you can fully define the service contract using gRPC's Protocol Buffer based Interface Definition Language (IDL) and then generate service and client code for your preferred programming language (Go, Java, Node, etc.).

GraphQL is also becoming quite popular for some use cases where you cannot have a fixed service contract. GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. From the foundation level itself, it is different from conventional client-server communication as it allows clients to determine what data they want, how they want it, and what format they want it in.

Thrift is used (where you can define an interface definition for your microservice), as an alternative to REST/HTTP synchronous messaging.

Most of the synchronous request-response style messaging is more suitable for services and clients that are interactive and requires real-time processing such as external facing APIs.

4.2 Asynchronous Messaging - Kafka, NATS, AMQP, STOMP, MQTT

The messaging between services can be based on event-driven messaging as well. With event-driven asynchronous messaging the client either doesn't expect a response immediately or at all. For such scenarios, microservices can leverage messaging protocols such as NATS, Kafka, AMQP, STOMP, and MQTT. Unlike most conventional messaging protocols, Kafka and NATS offer a dumb but distributed messaging infrastructure, which is more suitable for building microservices. This is so that producers and consumers will have all business logic while the broker doesn't have any.

4.3 Message Formats - JSON, XML, Thrift, ProtoBuf, Avro

Deciding the most suited message format for microservices is another key factor. Traditional monolithic applications use complex binary formats while SOA- and web services-based applications use text messages based on complex message formats (SOAP) and schemas (XSD). Most microservices based applications use simple text-based message formats, such as JSON and XML on top of HTTP REST APIs. In cases where we need binary message formats (text messages can become verbose in some use cases), microservices can leverage binary message formats, such as binary Thrift, ProtoBuf or Avro.

4.4 Service Contracts - Defining the Service Interfaces - OpenAPI/Swagger, gRPC IDL, GraphQL schema, Thrift IDL

When you have a business capability implemented as a service, you need to define and publish the service contract. In traditional monolithic applications, we barely find such features to define the business capabilities of an application. In the SOA/web services world, WSDL is used to define the service contract. But WSDL is not the ideal solution for defining a microservices contract as it does not deal with REST as a first-class citizen.

Since we build microservices on top of the REST architectural style, we can use the same REST API definition techniques to define the contract of the microservices. Therefore, microservices use the standard REST API definition languages, such as Swagger and RAML, to define the service contracts.

For other microservice implementations that are not based on HTTP/REST, such as gRPC and Thrift, we can use the protocol level IDL. With GraphQL, you can define service schema using GraphQL schema which can be used by the clients to query the GraphQL-based API.

5. Decentralized Data Management

In monolithic architecture the application stores data in single and centralized databases to implement various capabilities of the application.

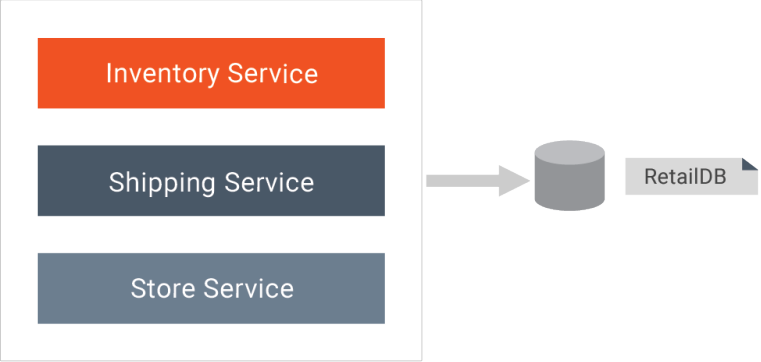

Figure 4: Monolithic application uses a centralized database to implement all its features

In MSA the business capabilities are dispersed across multiple microservices. So if we use the same centralized database, it's hard to ensure the loose coupling between services (for instance, if the database schema has changed from a given microservice, that will break several other services). Therefore, each microservice would need to have its own database.

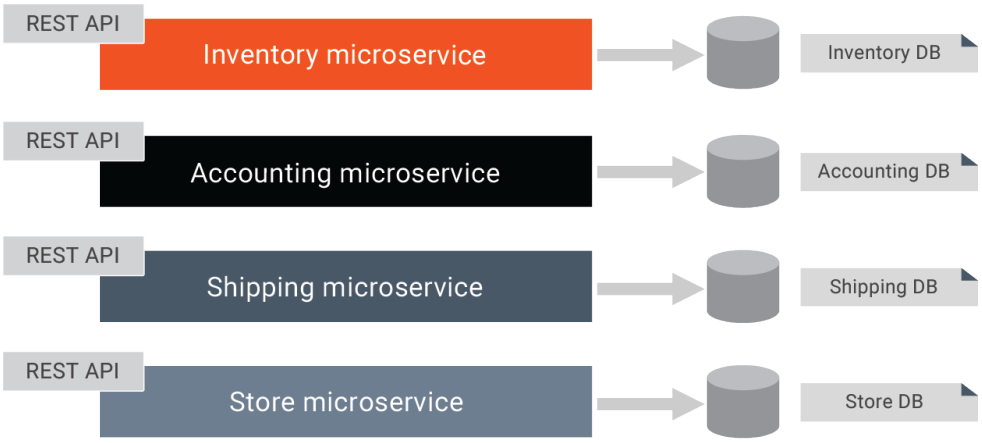

Figure 5: Microservices have its own private database and they can't directly access the database owned by other microservices

Here are the key aspect of implementing decentralized data management in MSA

- Each microservice can have a private database to persist the data that is required to implement the business functionality offered by it.
- A given microservice can only access the dedicated private database, but not the databases of other microservices.
- In some business scenarios, you might have to update several databases for a single transaction. In such scenarios, the databases of other microservices should be updated through its service API only (not allowed to access the database directly).

De-centralized data management will give you fully decoupled microservices and the liberty of choosing disparate data management techniques (SQL or NoSQL and different database management systems for each service). When you need synchronous messaging across multiple microservices, we can use several techniques. This includes building a service composition where one service calls multiple services to update the required database via the service API. We can also use event-based messaging between services to propagate data across multiple services and form different materialized views of the data. There are some related patterns such as Command Query Responsibility Segregation (CQRS) which is related to data management in microservices. You can find more information about microservice data management techniques and patterns in this book.

6. Governance

'Governance' in the context of IT is defined [3] as the processes that ensure the effective and efficient use of IT in enabling an organization to achieve its goals. SOA governance guides the development of reusable services, establishing how services will be designed and developed and how those services will change over time. It establishes agreements between the providers of services and the consumers of those services, telling the consumers what they can expect and the providers what they're obligated to provide. In SOA governance there are two types of governance that are commonly used:

- Design-time governance - defining and controlling the service creations, design, and implementation of service policies
- Run-time governance - the ability to enforce service policies during execution

So what does governance in the context of microservices really mean? There are a few discussions on positioning microservice governance as a fully decentralized process, but if we have a closer look at various aspects of microservices architecture, it's quite clear that

there are both centralized and decentralized aspect of microservices governance. In MSA, we can identify multiple aspects of governance:

- Development and lifecycle management
- Service registry and discovery
- Observability
- Service and API management

With respect to development lifecycle management, microservices are built as fully independent and decoupled services with a variety of technologies and platforms. Therefore, there is no need for defining a common standard for the design and development of services. We can summarize the decentralized governance capabilities of microservices as follows:

- Microservices can make their own decisions about the design and implementation
- MSA fosters the sharing of common/reusable services
- Run-time governances aspects, such as SLAs, throttling, monitoring, common security requirements, and service discovery, are not implemented at each microservice level. Rather, they are realized at a dedicated component (often at the API-gateway level)

Service registry and discovery is more or less centralized process where you keep track of services metadata in a central repository. Similarly, observability is also centralized where you publish your services metrics, logging and tracing related data to tools that operate in centralized mode across the board. When it comes to exposing a selected set of services as managed services or managed APIs, we can leverage API management techniques. API Management is also implemented as a centralized component in an MSA. We'll discuss all these governance aspects in detail in the upcoming sections.

7. Service Registry and Service Discovery

In MSA the number of microservices that you need to deal with is quite high. Their locations change dynamically too owing to the rapid and agile development/deployment nature of microservices. Therefore, you need to find the location of a microservice during the runtime. The solution to this problem is to use a service registry.

7.1 Service Registry

The service registry holds the metadata of microservice instances (which include its actual locations, host port, etc.). Microservice instances are registered with the service registry on startup and de-registered on shutdown. Consumers can find the available microservices and their locations through the service registry.

7.2 Service Discovery

To find the available microservices and their location, we need to have a service discovery mechanism. There are two types of service discovery mechanisms - client-side discovery and server-side discovery. Let's have a closer look at those service discovery mechanisms:

7.2.1 Client-side discovery

In this approach, the client or the API gateway obtains the location of a service instance by querying a service registry.

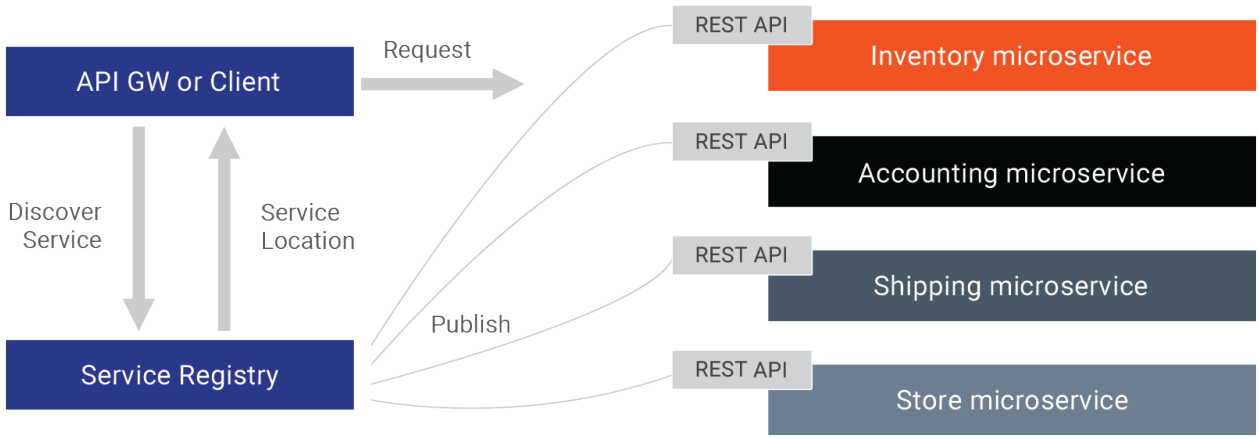

Figure 6: Client-side discovery

Here the client or API gateway has to implement the service discovery logic by calling the service registry component.

7.2.2 Server-side discovery

With this approach, the client or API gateway sends the request to a component (such as a load balancer) that runs on a well-known location. That component calls the service registry and determines the location of the requested microservice.

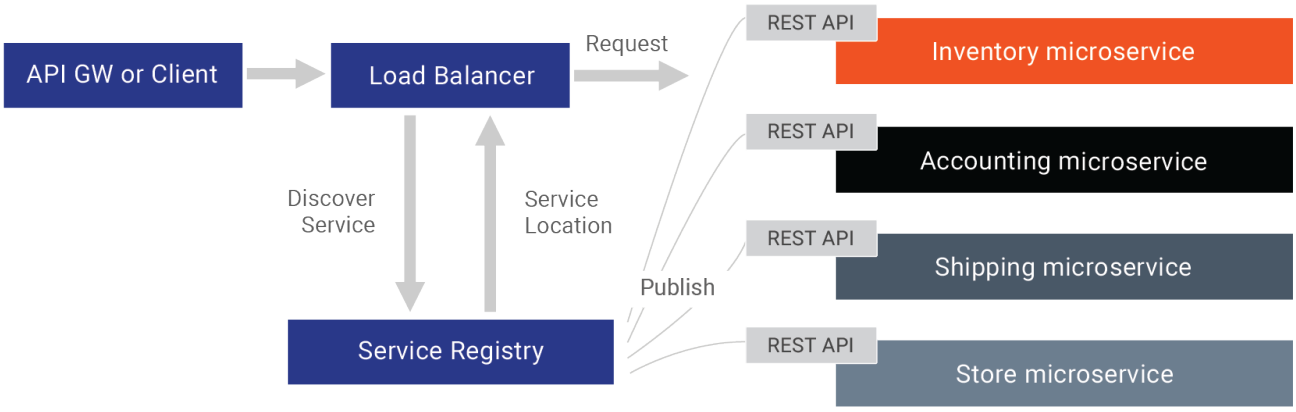

Figure 7: Server-side discovery

Microservices can leverage deployment solutions such as Kubernetes for server-side discovery.

Kubernetes offers built-in service registry and discovery capabilities so that you can call your service by its logical name and Kubernetes takes care of resolving them to actual IPs and ports.

8. Deployment

When it comes to MSA, the deployment of microservices plays a critical role and has the following key requirements:

- Ability to deploy/undeploy independently of other microservices
- Must be able to scale at each microservices level (a given service may get more traffic than other services)
- Deploying microservices quickly
- Failure in one microservice must not affect any of the other services

Docker (an open source engine that lets developers and system administrators deploy self-sufficient application containers in Linux environments) provides a great way to deploy

microservices while addressing the above requirements. The key steps involved are as follows:

- Package the microservice as a (Docker) container image
- Deploy each service instance as a container
- It is scaled according to the number of container instances
- Building, deploying, and starting a microservice will be much faster as we are using Docker containers (which are much faster than a regular VM)

Kubernetes extends Docker's capabilities by allowing to manage a cluster of Linux containers as a single system, managing and running Docker containers across multiple hosts, and offering co-location of containers, service discovery, and replication control. As you can see, most of these features are essential in the microservices context too. Hence, using Kubernetes (on top of Docker) for microservices deployment has become an extremely powerful approach, especially for large-scale microservices deployments.

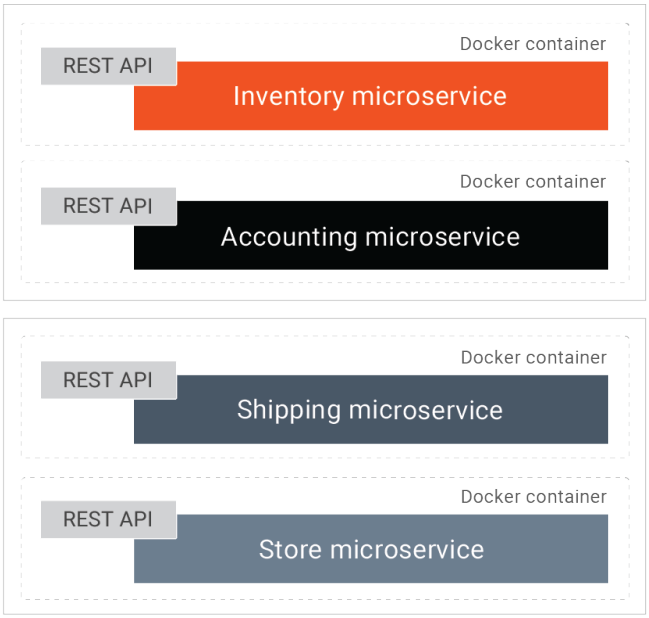

Figure 8: Building and deploying microservices as containers

Figure 8 shows how the microservices are deployed in the retail application. Each microservice instance is deployed as a container and there are two containers per host.

9. Security

Securing microservices is quite a common requirement when you used in real-world scenarios. Before jumping into microservices security let's have a quick look at how we normally implement security at the monolithic application level:

- In a typical monolithic application, security is about finding 'who is the caller', 'what can the caller do' and 'how do we propagate that information'.
- This is usually implemented at a common security component, which is at the beginning of the request handling chain. That component populates the required information with the use of an underlying user repository (or user store).

So, can we directly translate this pattern into the MSA? Yes, but that requires a security component implemented at each microservices level that’s talking to a centralized/shared user repository to retrieve the required information. That's a very tedious approach to solving the microservices security problem.

Instead, we can leverage widely used API-security standards, such as OAuth 2.0 and OpenID Connect (OIDC), to find a better solution. Before deep-diving into that, let’s first summarize the purpose of each standard and how we can use them.

- OAuth 2.0 is an access delegation protocol. The client authenticates with an authorization server and gets an opaque token, which is known as the 'access token'. The access token has zero information about the user or client. It only has a reference to the user information, which can only be retrieved by the authorization server. Hence, this is known as a 'by-reference token' and is safe to use this token even in a public network or the Internet.
- OIDC behaves similar to OAuth 2.0, but in addition to the access token, the authorization server issues an ID token that contains information about the user. This is often implemented by a JSON Web Token (JWT) that is signed by the authorization server. This ensures the trust between the authorization server and the client. JWT is therefore known as a 'by-value token' as it contains the information of the user and isn’t safe to use outside the internal network.

Now, let's see how we can use these standards to secure microservices in our retail example.

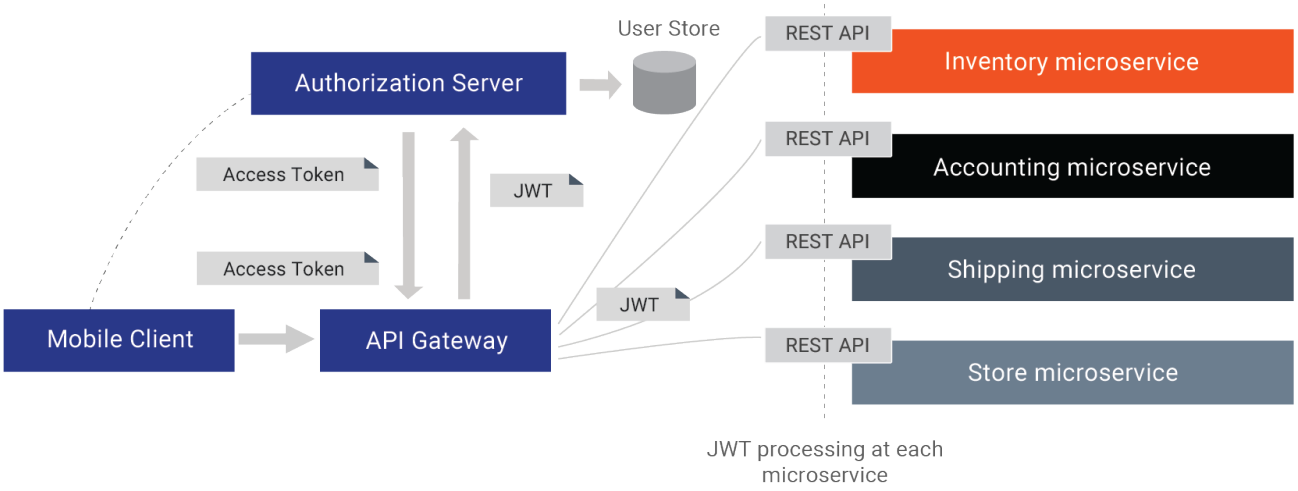

Figure 9: Microservice security with OAuth 2.0 and OpenID Connect

As shown in Figure 9, these are the key steps involved in implementing microservices security:

- Leave authentication to OAuth 2.0 and OIDC server (authorization server). Access to the microservices will be successfully provided given that someone has the right to use the data.
- Use the API gateway style in which there is a single entry point for all client requests.
- The client connects to the authorization server and obtains the access token (by-reference token). Then sends the access token to the API gateway along with the request.
- Token translation at the gateway - API gateway extracts the access token and sends it to the authorization server to retrieve the JWT (by value-token).
- The gateway passes this JWT along with the request to the microservices layer.
- JWTs contain the necessary information to help in storing user sessions and other data. If each service can understand a JWT, then you have distributed the identity mechanism, which allows you to transport identity throughout your system.
- At each microservice layer, we can have a component that processes the JWT, which is quite a trivial implementation.

10. Design for Failures

MSA introduces a dispersed set of services and, in comparison with monolithic design, it increases the possibility of having failures at each service level. In fact, all these technologies are not really invented along with MSA, but have been in the software application development space for quite some time (MSA merely emphasized the importance of those concepts).

A given microservice can fail due to network issues and unavailability of underlying resources among other things. An unavailable or unresponsive microservice should not bring the whole microservices-based application down. Thus, microservices should be fault tolerant and be able to recover when possible. Additionally, the client has to handle it gracefully.

Moreover, since services can fail at any time, it's important to be able to detect (real-time monitoring) the failures quickly and, if possible, automatically restore these services.

There are several commonly used patterns in handling errors in the context of microservices.

10.1 Circuit Breaker

When you’re doing an external call to a microservice, you configure a fault monitor component with each invocation. When the failures reach a certain threshold that component stops any further invocations of the service (trips the circuit). After a certain number of requests in open state (which you can configure), change the circuit back to close state.

This pattern is quite useful to avoid unnecessary resource consumption and request delays due to timeouts. It also gives us the ability to monitor the system (based on the active open circuits states).

10.2 Bulkhead

Given that a microservice application comprises a number of microservices, the failure of one part of the microservices-based application should not affect the rest of the application. Bulkhead pattern is about isolating different parts of your application so that a failure of a service in a part of the application does not affect any of the other services.

10.3 Timeout

The timeout pattern is a mechanism that allows you to stop waiting for a response from the microservice when you think it won't come. Here, you can configure the time interval you wish to wait.

The patterns that we discussed above are commonly used in inter-microservice communication. Most of these patterns are available as libraries (e.g. Hystrix) for different programming languages and you can simply reuse them in the services that you develop.

11. Inter-Service/Process Communication

In MSA, the software applications are built as a suite of independent services. Therefore, in order to realize a business use case, you need to have communication structures between different microservices/processes. That's why inter-service/process communication between microservices is a vital aspect.

In SOA implementations, the inter-service communication between services is facilitated by a central runtime know as the Enterprise Service Bus (ESB) and most of the business logic resides in that intermediate layer (message routing, transformation, and service orchestration). However, MSA eliminates the central message bus/ESB and moves the 'smartness' or business logic to the services and client (known as 'smart endpoints'). Therefore the business logic and the network communication logic that is required to call other services and systems is implemented as part of the microservice itself.

The microservice interactions or integrations will be built based on two main integration styles: Active Composition and Reaction Composition.

11.1 Active Composition

If we have a closer look at the microservices implementation, we can identify different types of services as shown in figure 10.

Figure 10: Active composition of microservices

We have fine-grained self-contained services (no external service dependencies) that mostly comprise of the business logic and less or no network communication logic. We can categorize them as atomic/core services.

Atomic/core microservices often cannot be directly mapped to a business functionality as they are too fine-grained. Hence a specific business functionality may require a composition of multiple atomic/core services. A given microservice can invoke multiple downstream services with a synchronous request-response messaging style and create a composite service. Such a composition is known as an active composition. These services are often called composite or integration services where a significant portion of the ESB functionality that we had in SOA such as routing, transformations, orchestration, resilience, and stability patterns are implemented.

The business functionality is exposed to the consumers as managed APIs and a selected set of your composite services or even some atomic service will be exposed as managed APIs using API services/edge services. These services are a special type of composite services, that apply basic routing capabilities, versioning of APIs, API security patterns, throttling, monetization, and creation of API compositions among other things.

11.2 Reactive Composition

With the active composition style, the composite services cannot fully operate autonomously. While such services are good for interacting with API or external facing services, most of the internal business logic of microservices-based applications can be implemented using asynchronous event-driven communication between the services.

This style of building inter-service communications is known as reactive composition. As shown in figure 11, microservices can use a centralized or decentralized event bus/broker which acts as the 'dumb pipe' and all the smarts live at the producer microservices and consumer microservices.

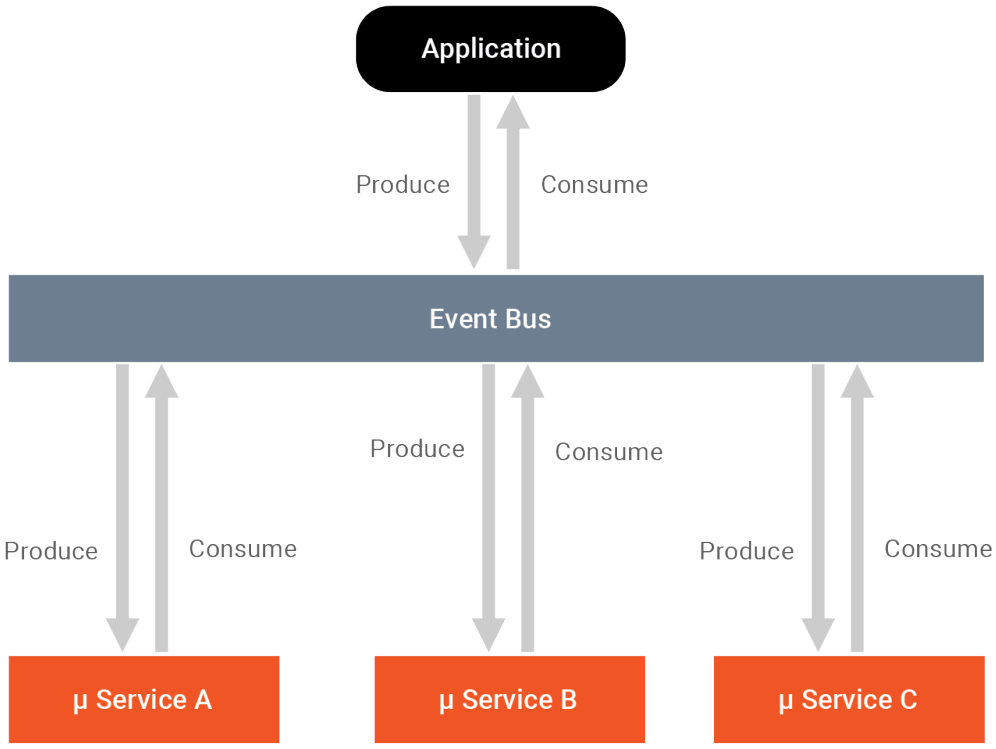

Figure 11: All microservices are exposed via an API-gateway

The event bus can often be implemented with technologies such as Kafka, AMQP, and NATS.io. Depending on the use case you can select an in-memory or persistent layer to back the event bus.

11.3 Hybrid Composition

In most pragmatic applications of microservices, active and reactive composition models are used in a hybrid manner. As shown in figure 12, you can build most of the interactive and external facing services in active style while the internal service communication which requires different delivery guarantees can be implemented in a reactive style.

Figure 12: Hybrid composition

The API layer usually sits above the composition layer and other external and monolithic subsystems can also be integrated through the composition layer.

11.4 API Management

Microservices can be exposed via the gateway and all API management techniques can be applied at that layer. All other requirements such as security, throttling, caching, monetization, and monitoring have to be done at the gateway layer. The API gateway layer can often be segregated into multiple gateway instances (often known as a micro gateway which is assigned per API) while API management components remain central. It is important to minimize [1] the business logic that you put at the API gateway layer.

12. Service Mesh

Implementing the functionality related to service-to-service communication from scratch is a nightmare. Rather than focusing on the business logic, you will have to spend a lot of time building service-to-service communication functionality. This is even worse if you use multiple technologies to build microservices because you need to duplicate the same effort across different languages (e.g. circuit breaker has to be implemented on Java, Node, or Python).

Since most of the inter-service communication requirements are quite generic across all microservices implementations, we can think about offloading all such tasks to a different layer, so that we can keep the service code independent. That's where 'service mesh' comes into the picture.

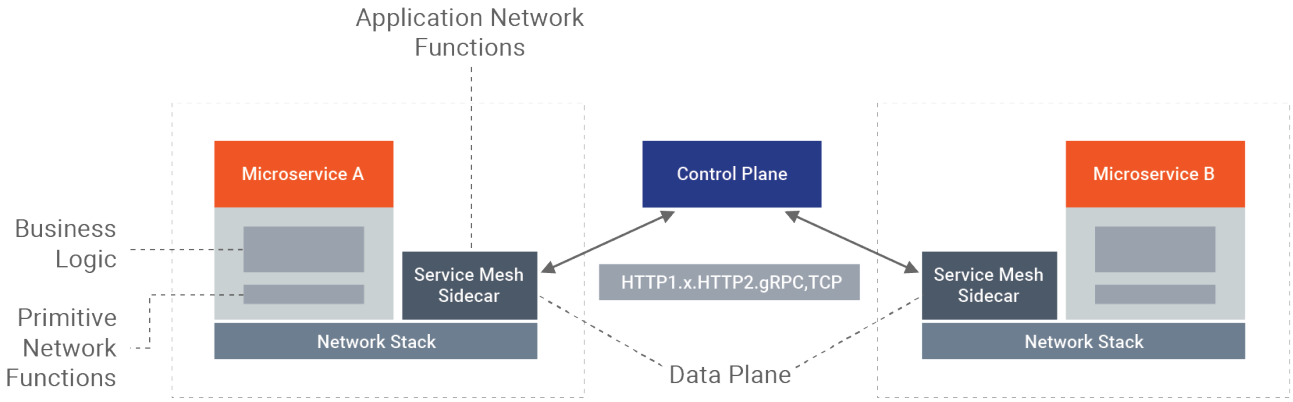

Figure 13: Service Mesh in Action

A service mesh is an inter-service communication infrastructure. This means that a given microservice won't directly communicate with the other microservices. All service-to-service communications will take place on top of a software component called the service

mesh (or side-car proxy). The service mesh provides built-in support for network functions such as resiliency and service discovery. Therefore, service developers can focus more on business logic while most of the work related to network communication is offloaded to the service mesh. For instance, you don't need to worry about circuit breaking when your microservice calls another service anymore. That already comes as part of the service mesh. Service mesh is language agnostic. Since the microservice to service mesh proxy communication is always on top of standard protocols such as HTTP1.1/2.x, and gRPC, you can write your microservice from any technology and it will still work with the service mesh.

This is the key functionality offered by a service mesh:

- Resiliency for inter-service communications: Circuit-breaking, retries and timeouts, fault injection, fault handling, load balancing, and failover
- Service discovery: Discovery of service endpoints through a dedicated service registry
- Routing: Primitive routing capabilities, but no routing logic related to the business functionality of the service
- Observability: Metrics, monitoring, distributed logging, and distributed tracing
- Security: Transport level security (TLS) and key management
- Access control: Simple blacklist and whitelist based access control
- Deployment: Native support for containers. Docker and Kubernetes
- Inter-service communication protocols: HTTP1.1, HTTP2, gRPC

It's important to understand that service mesh is completely independent of your service's business logic. You can consider it as a network abstraction. You are responsible for implementing the business functionality of your service. Therefore, it is by no means a distributed ESB (refer to [1] and [2] for more details on this topic).

13. Transactions

What about transactions support in microservices? In fact, supporting distributed transactions across multiple microservices is a complex task. The microservice architecture itself encourages transaction-less coordination between services.

The idea is that a given service is fully self-contained and based on the single responsibility principle. Hence, in most cases, transactions are applicable only at the scope of the microservices (i.e. not across multiple microservices).

However, if there's a mandatory requirement to have distributed transactions across multiple services, we should avoid two-phase-commit (2PC) at all cost. Such scenarios can be realized with the introduction of the SAGA pattern [1] which involves using 'compensating operations' at each microservice level. The key idea is that a given microservice is based on the single responsibility principle and if a given microservice fails to execute a given operation, we can consider that as a failure of that entire microservice. Then all the other (upstream) operations have to be undone by invoking the respective compensating operation of those microservices. You can refer to [1] for more details on realizing the SAGA pattern to build transactions between microservices.

14. Realizing MSA with WSO2

You can leverage WSO2's cloud native and 100% open source technology to implement different aspects of an MSA.

For building microservices, you can leverage the Ballerina programming language which is powered by WSO2. Ballerina is a cloud native programming language that is designed to make the development of distributed applications simple. It natively offers abstractions for network interactions, network types, resilient inter-service communication, data integration, observability and integration with cloud native ecosystem. Therefore Ballerina is ideal for developing services that create a composition (active or reaction) of multiple microservices (core services).

You can also use the WSO2 Micro Integrator, which is a cloud native runtime that allows you to integrate microservices using active or reactive composition patterns based on an intuitive graphical development tool or using a configuration language (DSL). WSO2 Micro Integrator is a variant of the proven and battle-tested WSO2 Enterprise Integrator/WSO2 ESB. So depending on your preference you can select either Ballerina or Micro Integrator for building composite services (figure 13). Also, you can use a central WSO2 Enterprise Integrator component to integrate with the existing monolithic subsystems.

API service layer can be implemented with WSO2 API Manager and WSO2 Identity Server can be used as the identity provider and key manager for securing your API traffic. All the WSO2 technologies seamlessly integrate with deployment technologies such as Docker, Kubernetes, and Istio service mesh, and observability tools such as Prometheus, Grafana, Zipkin, and Jaeger.

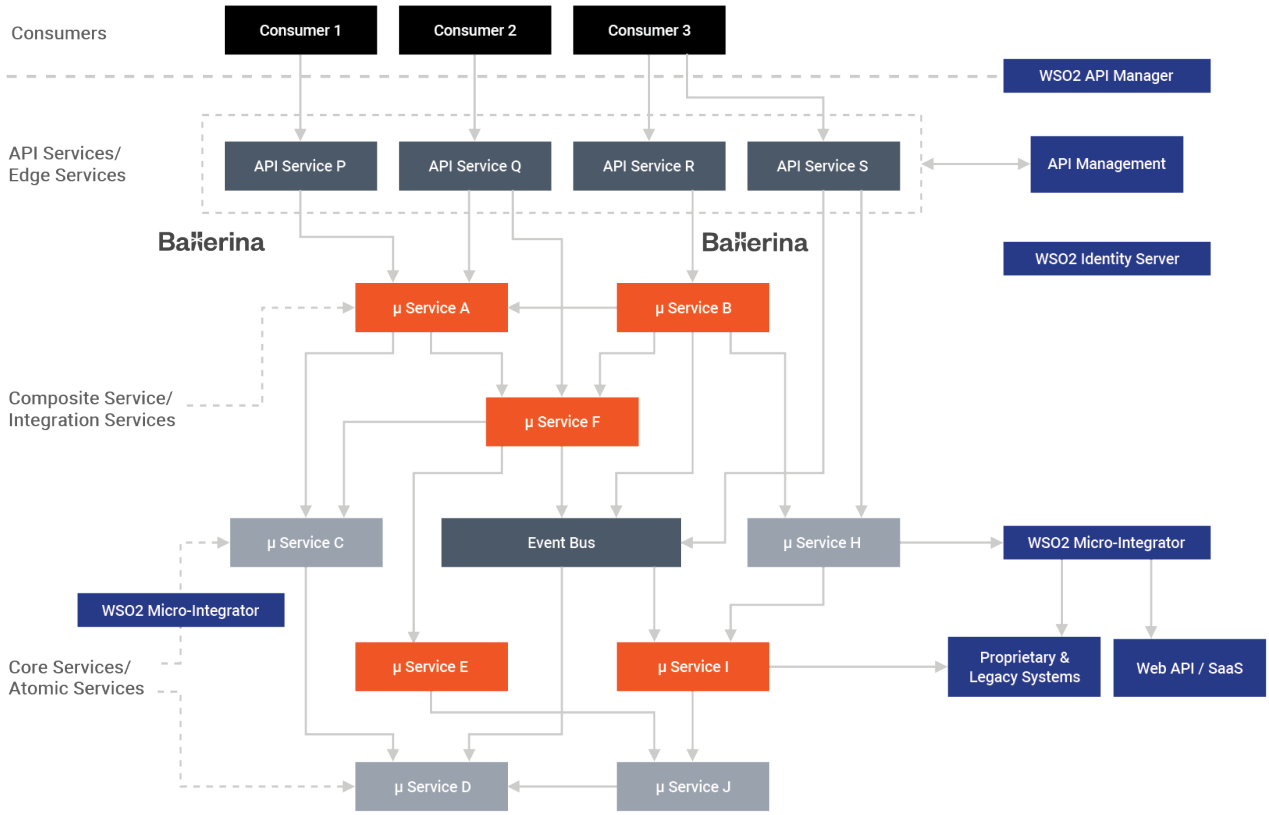

Figure 14: Realizing microservices architecture with WSO2 products and technologies.

15. Conclusion

When determining how you can incorporate an MSA in today's modern enterprise IT environment, we can summarize the following key aspects:

- Microservices is not a panacea - it won't solve all your enterprise IT needs, so we need to use it with other existing architectures.
- It's pretty much SOA done right with the addition of containers and container orchestration (Kubernetes).
- Most enterprise won't be able to convert their entire enterprise IT systems to microservices. Instead, they will use microservices to address some business use cases where they can leverage the power of MSA.
- Enterprise integration will never go away - There won't be a central integration layer to integrate your services but the integration requirements never go away. But rather they are now being implemented at each microservice level. The monolithic subsystems will still require integration in the form of a central bus/ESB. So a hybrid or coexistence of both architectural styles is more pragmatic.

- All business capabilities should be exposed as APIs by leveraging API management techniques.
- You can offload some of the network communication-related complexity to the service mesh layer but not the business logic.
- Diversification of the technology choices and selecting the best of breed technologies is essential to build your microservices.

16. References

[1] "Microservices for the Enterprise"

[2] "Application Integration for Microservices Architectures: A Service Mesh Is Not an ESB"

[3] "https://martinfowler.com/articles/microservices.htm"