WHITE PAPER

Microservices in Practice - Key Architectural Concepts of an MSA

By Kasun Indrasiri
Director - Integration Technologies, WSO2

1. Monolithic Architecture

Enterprise software applications are designed to facilitate numerous business requirements. Hence, a given software application offers hundreds of functions and all such functions are generally piled into a single monolithic application. ERP, CRM, and other various software systems are good examples - they're built as monoliths with several hundreds of functions. The deployment, troubleshooting, scaling, and upgrading of such software applications are 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 promote a software application to be implemented as a monolith, which comprises 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 functions. 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 multiple services. All these services are deployed into the same application runtime. Therefore, it shows several characteristics of a monolithic application: it's complex, designed, developed, and deployed as a single unit; it's hard to practice agile development and delivery methodologies; updating a part of the application required 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. one service requiring 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 microservice architecture coming into being. 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.

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 are not just about splitting the services available in monolith into independent services.

Consider that by looking at the functions offered from the monolith, by identifying the business capabilities required from the application - that is 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 and 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 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's 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 microservice architectural 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 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 functions to several dozens of functions. Having webservices and rebranding them as microservices is not going to give you any benefits of MSA.

So, then how should we properly design services in an MSA?

3.1 Guidelines for Designing Microservices

- Single Responsibility Principle (SRP): Having a limited and a focused business scope for a microservice helps us to meet the agility in development and delivery of services.
- During the designing phase of the microservices, we should find their boundaries and align them with the business capabilities (also known as bounded context in DomainDriven-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, but not about making the service smaller.
- It is often a good practice to start with relatively broad service boundaries to begin with, refactoring to smaller ones (based on business requirements) as time goes on.

In our retail use case, you can find that we have split the functions of its monolith into four different microservices namely 'inventory', 'accounting', 'shipping' and 'store'. They are addressing a limited, but focussed 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 functions 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, JMS.

4.1 Synchronous Messaging - REST, Thrift

For synchronous messaging (the client expects a timely response from the service and waits to get it) in MSA, REST is the unanimous choice as it provides a simple messaging style implemented with HTTP request-response, based on resources. Therefore, most microservice implementations are using HTTP along with resources (every functionality is represented with a resource and operations carried out on top of those resources).

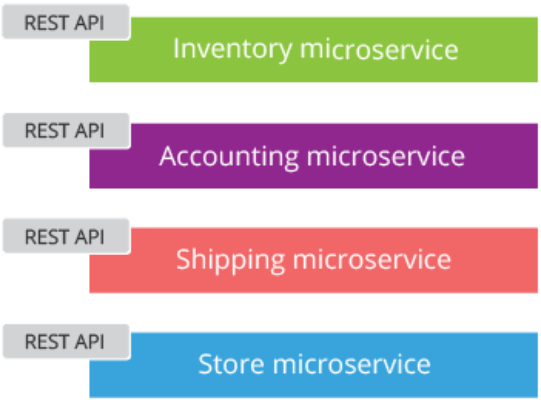

Figure 3: Using REST interfaces to expose microservices

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

4.2 Asynchronous Messaging - AMQP, STOMP, MQTT

For some microservice scenarios, it is required to use asynchronous messaging techniques (the client doesn't expect a response immediately, or not at all). In such scenarios, asynchronous messaging protocols, such as AMQP, STOMP or MQTT, are widely used.

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

Deciding the most suited message format for microservices is a another key factor. The traditional monolithic applications use complex binary formats, SOA/Web services-based applications use text messages based on the 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 API. 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 - Swagger, RAML, 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 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 Thrift, we can use the protocol level 'Interface Definition Languages (IDL)' (e.g. Thrift IDL).

5. Decentralized Data Management

3. Decentralized Data Management

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

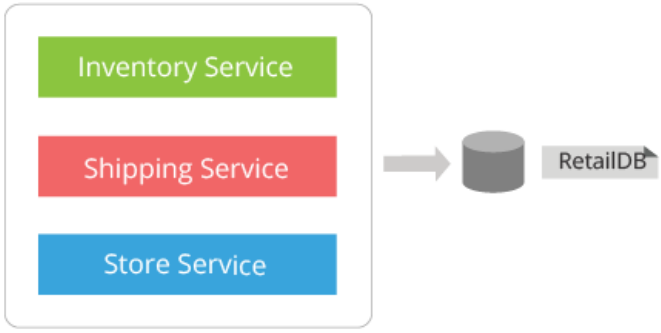

Figure 4: Monolithic application uses a centralized database to implement all its features

In MSA the functions are dispersed across multiple microservices and 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.

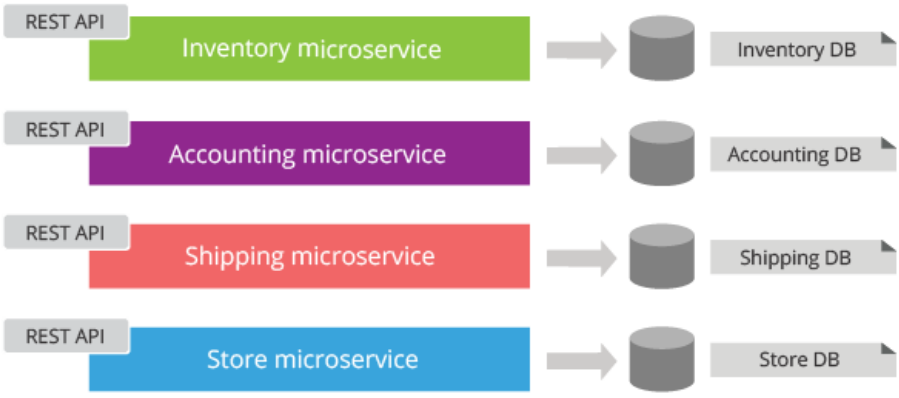

Figure 5: Microservices have their own private database and they can't directly access the database owned by other microservices

Here are the key aspects of implementing decentralized data management in MSA.

- Each microservice can have a private database to persist the data that requires to implement the business functionality offered from 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).

The de-centralized data management will give you fully decoupled microservices and the liberty of choosing disparate data management techniques (SQL or NoSQL etc., different database management systems for each service). However, for complex transactional use cases that involve multiple microservices, the transactional behavior has to be implemented using the APIs offered from each service and the logic resides either at the client or intermediary (GW) level.

6. Decentralized Governance

MSA favors decentralized 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. In the context of SOA, 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 in common use:

- 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? In MSA, microservices are built as fully independent and decoupled services with the variety of technologies and platforms. Therefore, there is no need of defining a common standards for services designing and development. We can summarize decentralized governance capabilities of microservices as follows:

- Microservices can make their own decisions about design and implementation.
- MSA fosters the sharing of common/reusable services.
- Run-time governance 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).

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 runtime. The solution to this problem is to use a Service Registry.

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.

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:

- Client-side discovery

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

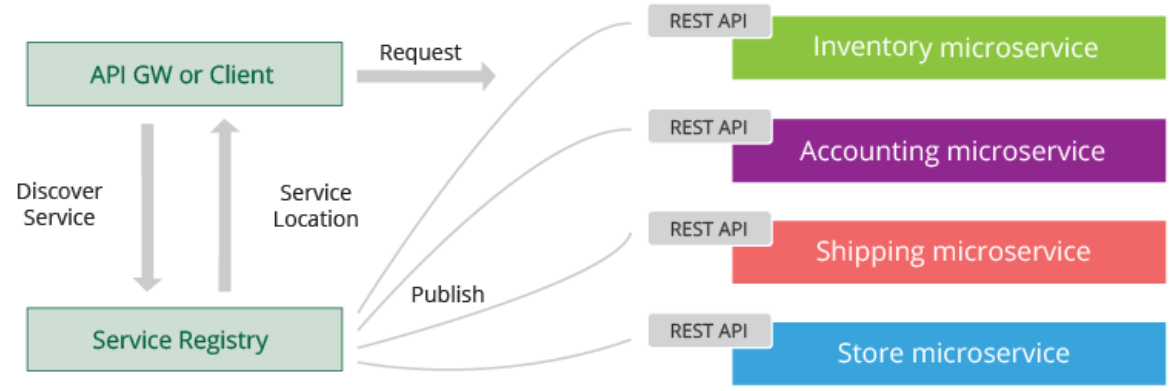

Figure 6: Client-side discovery

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

- Server-side discovery

With this approach, clients/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.

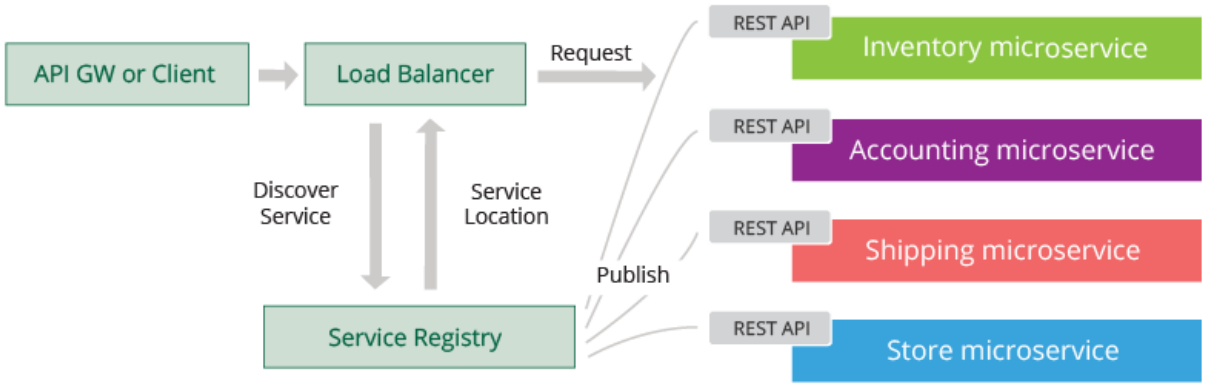

Figure 7: Server-side discovery

The microservices can leverage the deployment solutions, such as Kubernetes for service-side discovery.

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 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
- Scaling is done based on changing the number of container instances.
- Building, deploying, and starting a microservice will be much faster as we are using Docker containers (which is much faster than a regular VM).

Kubernetes is extending Docker's capabilities by allowing to manage a cluster of Linux containers as a single system, managing and running Docker containers across multiple hosts, 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.

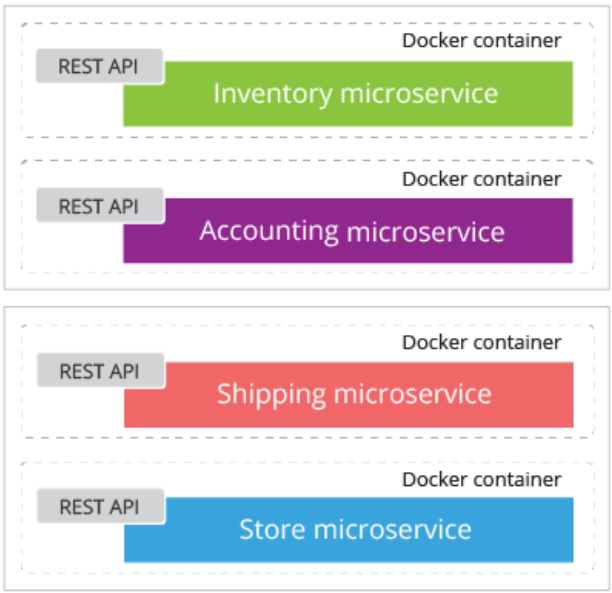

Figure 8: Building and deploying microservices as containers

Figure 8 shows an overview of the deployment of microservices of 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 use microservices in realworld scenarios. Before jumping in to 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 that '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 and 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 and retrieve the required information. That's a very tedious approach of solving the microservices security problem.

Instead, we can leverage the widely used API-security standards, such as OAuth 2.0 and OpenID Connect, to find a better solution to the microservices security problem. 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/client. It only has a reference to the user information that can only be retrieved by the authorization server. Hence, this is known as a 'by-reference token' and it is safe to use this token even in the public network/internet.
- OpenID Connect behaves similar to OAuth, 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 JWT (JSON Web Token) and that is signed by the authorization server. This ensures trust between the authorization server and the client. JWT is therefore known as a 'by-value token' as it contains information of the user and obviously is not safe to use outside the internal network.

Now, lets see how we can use these standards to secure microservices in our retail example

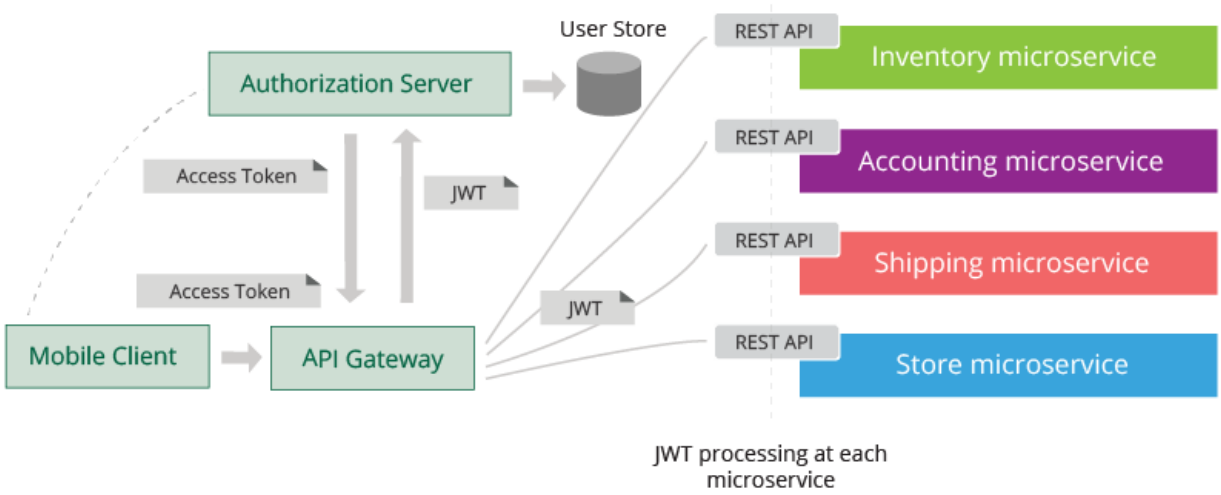

Figure 9: Microservice security with OAuth 2.0 and OpenID Connect

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

- Leave authentication to OAuth 2.0 and the OpenID Connect server (authorization server), so microservices successfully provide access 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.
- Client connects to the authorization server and obtains the access token (byreference token). Then send 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 contains the necessary information to help in storing user sessions, etc. If each service can understand a JSON web token, then you have distributed your identity mechanism that's allowing 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. Inter-Service/Process Communication

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

In SOA implementations, inter-service communication between services is facilitated by an enterprise service bus (ESB) and most of the business logic resides in the intermediate layer (message routing, transformation and orchestration). However, MSA promotes to eliminate the central message bus/ESB and move the 'smartness' or business logic to the services and client (known as 'smart endpoints').

Since microservices use standard protocols, (such as HTTP) and message formats, (such as JSON etc.) the requirement of integrating with a disparate protocol is minimal when it comes to communication among microservices. Another alternative approach in microservice communication is to use a lightweight message bus or gateway [2] with minimal routing capabilities and just acting as a 'dumb pipe' with no business logic implemented on the gateway. Based on these styles there are several communication patterns that have emerged in MSA.

10.1 Point-to-Point Style - Invoking Services Directly

In the point-to-point style, the entire message routing logic resides on each endpoint and the services can communicate directly. Each microservice exposes a REST APIs and a given microservice or an external client can invoke another microservice through its REST API.

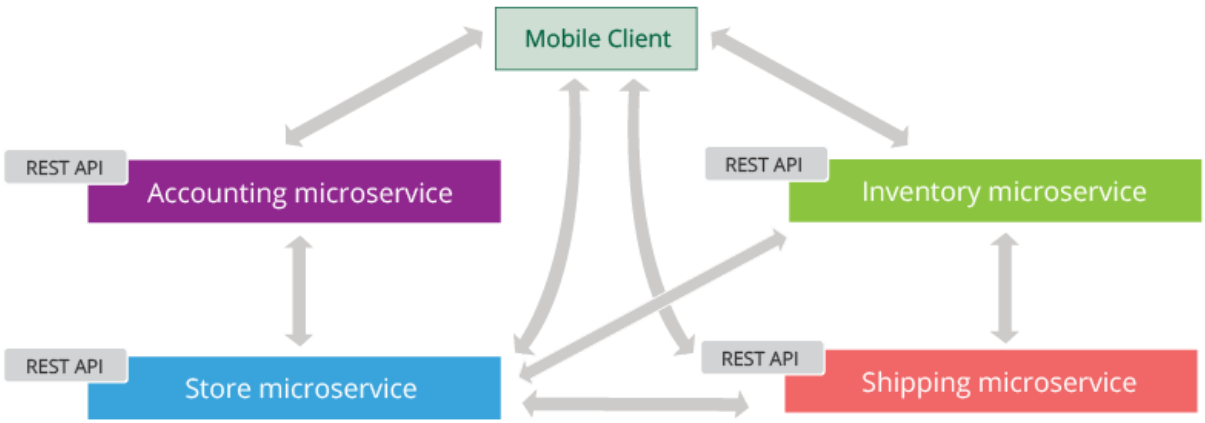

Figure 10: Inter-service communication with point-to-point connectivity

Obviously this model works for relatively simple microservices based applications, but as the number of services increases, this will become overwhelmingly complex. After all that's the exact same reason for using an ESB in the traditional SOA implementation, which is to get rid of this messy point-to-point integration links. Let's try to summarize the key drawbacks of the point-to-point style for microservice communication.

- The non-functional requirements, such as end-user authentication, throttling, monitoring, etc. needs to be implemented at each and every microservice level.
- As a result of duplicating common functions, each microservice implementation can become complex.
- No central place to apply non-functional requirements, such as monitoring, tracing, or security needs to be applied at each service level.
- Often the direct communication style is considered as a microservice anti-pattern for large-scale microservice implementations.

Therefore, for complex microservice use cases, rather than having point-to-point connectivity or a central ESB, we could have a lightweight central messaging bus that can provide an abstraction layer for the microservices and that can be used to implement various non-functional capabilities. This style is known as the API gateway style.

10.2 API-Gateway Style

The key idea behind the API gateway style is to use a lightweight message gateway as the main entry point for all clients/consumers and implement the common non-functional requirements at the gateway level. In general, an API gateway allows you to consume a managed API over REST/HTTP. As a result, you can expose your business functions that are implemented as microservices via the API-gateway as managed APIs. In fact, this is a combination of MSA and API management.

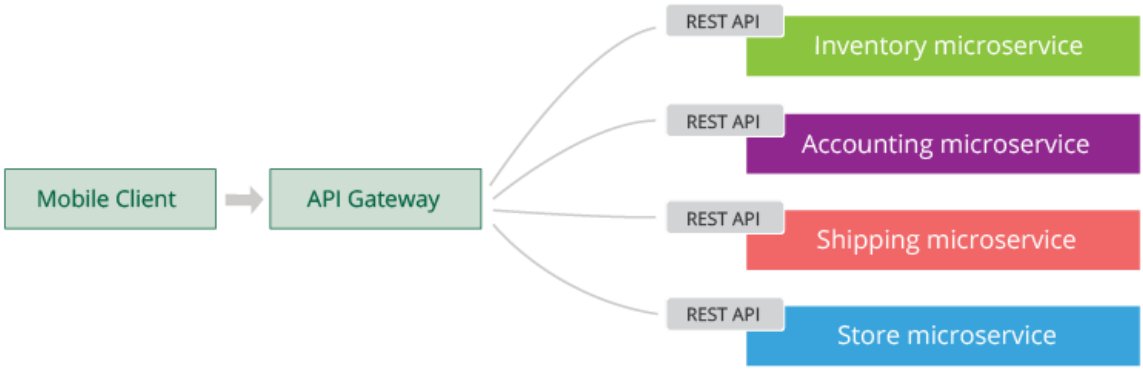

Figure 11: All microservices are exposed via an API-gateway

In our retail business scenario, as depicted in Figure 11, all the microservices are exposed via an API-gateway and that is the single entry point for all clients. If a microservice wants to consume another microservice that too needs to be done through the API-gateway.

The API-gateway style will give you the following advantages:

- Ability to provide the required abstractions at the gateway level for existing microservices, e.g. rather than providing a one-size-fits-all style API, the API-gateway can expose a different API for each client.
- Lightweight message routing/transformation at the gateway level - you can do basic message filtering, routing, and transformation at the gateway layer.
- Central place to apply non-functional capabilities, such as security, monitoring and throttling - rather implementing these generic features at each microservice layer.
- With the use of the API-gateway pattern, the microservice will become even more lightweight as all non-functional requirements are implemented at the gateway level.

The API-gateway style could well be the most widely-used pattern in most microservice implementations [3].

10.3 Message Broker Style

The microservices can be integrated in asynchronous messaging scenarios, such as one-way requests and publish-subscribe messaging using queues or topics. A given microservice can be the message producer and it can asynchronously send messages to a queue or topic. Then the consuming microservice can consume messages from the queue or topic. This style decouples message producers from message consumers and the intermediate message broker will buffer messages until the consumer is able to process them. Producer microservices are

completely unaware of the consumer microservices.

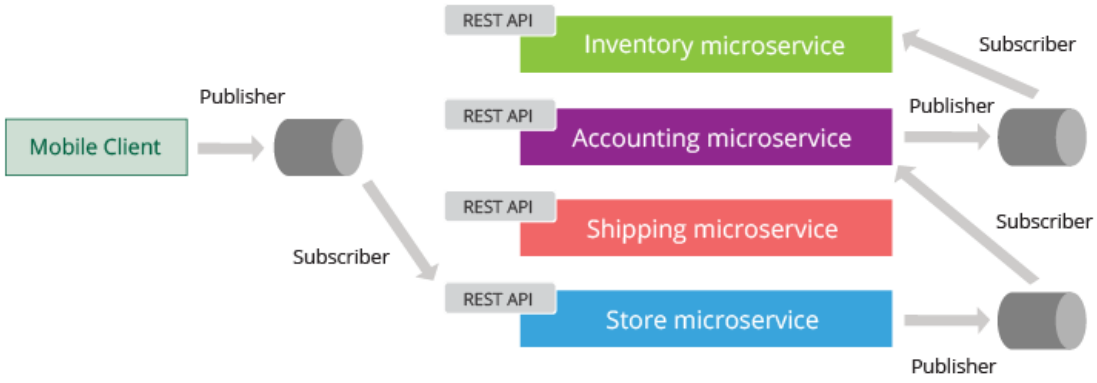

Figure 12: Asynchronous messaging based integration using pub-sub

Communication between consumers/producers is facilitated through a message broker that's based on asynchronous messaging standards, such as AMQP, MQTT, etc.

We will discuss about the interservice communication and how you can integrate microservices in the Integrating Microservices section.

11. 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, then such scenarios can be realized with the introduction of '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 failed to execute a given operation, we can consider that as a failure of that entire microservice. Then all other (upstream) operations have to be undone by invoking the respective compensating operation of those microservices.

12. Design for Failure

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 a some time (MSA merely emphasizes the importance of those concepts).

A given microservice can fail due to network issues, unavailability of underlying resources, etc. An unavailable or unresponsive microservice should not bring the whole microservicesbased application down. Thus, microservices should be fault tolerant, be able to recover when that's possible, and 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.

Circuit Breaker

When you're doing an external call to a microservice, you configure a fault monitor component with each invocation and 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, request delays due to timeouts, and also give us a chance to monitor the system (based on the active open circuits states).

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 such part of the application does not affect any of the other services.

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.

So, where and how do we use these patterns with microservices? In most cases, these patterns are applicable at the gateway level. This means when the microservices are not available or not responding at the gateway level, we can decide whether to send the request to the microservice using the circuit breaker or timeout pattern. It’s also important to have patterns such as bulkhead implemented at the gateway level as it’s the single entry point for all client requests, hence a failure in a given service should not affect the invocation of other microservices.

In addition, gateway can be used as the central point at which we can obtain the status and monitor each microservice, as each microservice is invoked through the gateway.

13. Microservices in Modern Enterprise Architecture

While MSA removes a lot of complexity from the service layer (when it comes to design, development and deployment), the complexity that's removed from the service layer has to be fulfilled by some other component/layer. For example, since MSA does not recommend the use of an ESB as the centralized bus, all tasks done by an ESB, such as service orchestration, routing, and integration with disparate systems must be done by other components, including microservices themselves.

MSA encourages the enterprise to build all of its IT solutions as microservices and not use any intermediate integration products, such as ESB. However, unless you’re a greenfield startup with no proprietary or legacy systems, that’s not a realistic approach. If you consider any large organization or a cooperation, you simply can’t convert all your software systems, services, and solutions to microservices. However, such organizations want to leverage MSA to build software solutions. Therefore, what we really need to have is a mix of MSA blended with the conventional architecture of existing systems.

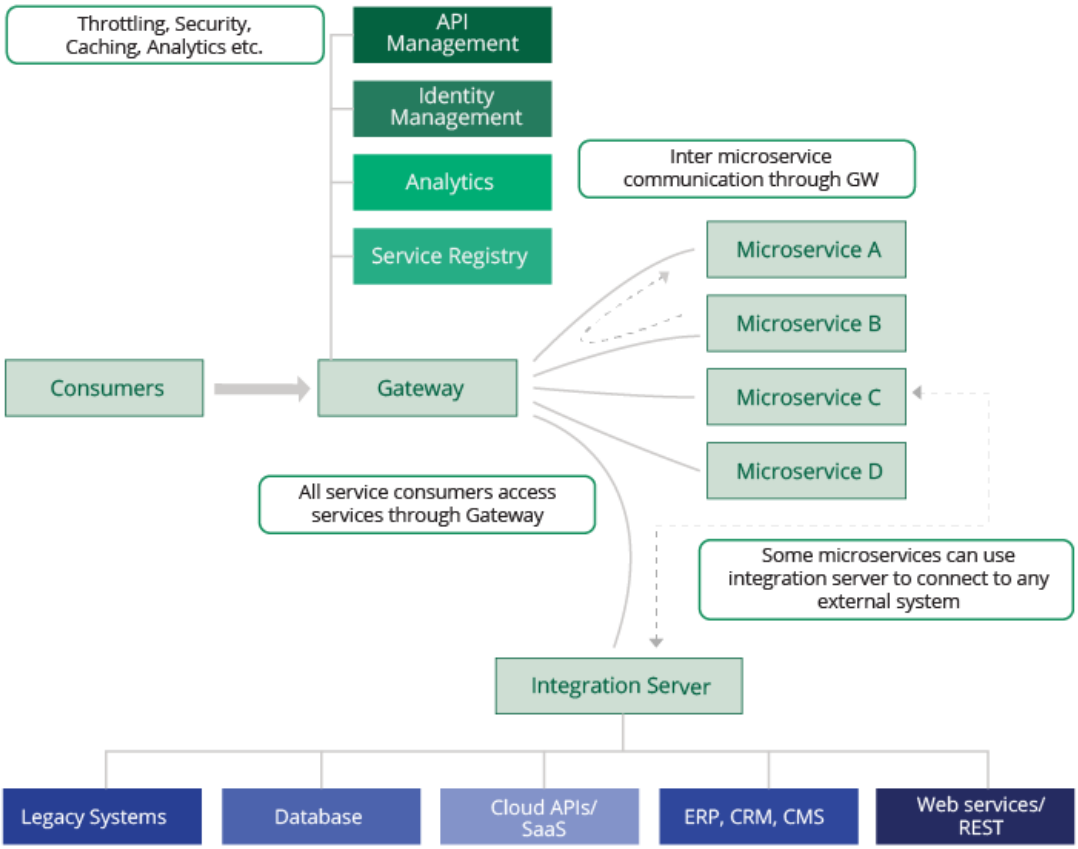

Figure 13: The modern enterprise architecture with microservices, enterprise integration, and API management

Figure 13 illustrates a high level enterprise IT architecture. Here we've used a hybrid architecture that comprises both microservices and existing systems. There are key design decisions that you need to take when you introduce MSA to your organization.

Here are the key aspects of introducing an MSA to your enterprise.

- Use MSA to build solutions whenever required and try to gain the full power that MSA brings in.
- Enterprise integration is still required; since we're going for a hybrid approach, you will still need to integrate all your internal systems and services with the use of an integration software, such as an ESB.
- You won't be able to throw away most of the existing systems, but the new microservices may need to call such monolithic systems to facilitate the various business requirements. In this case, you may use the underlying integration software/ ESB and the microservice can call the integration server to connect to disparate systems.
- The 'new' ESBs: Although integration software, such as ESB, may still be needed for the modern enterprise architecture, such tools won't be the central integration bus any more. An organization should look for lightweight, high-performant, and scalable integration software instead of heavyweight integration frameworks.
- API management: Microservices can be exposed via the gateway and all APImanagement techniques can be applied at that layer. All other requirements such as security, throttling, caching, monetization, and monitoring has to be done at the gateway layer. Moreover, the non-microservice based services (traditional SOA) can also be exposed through the API gateway.

14. Integrating Microservices

The most commonly asked questions in the microservice space are 'can microservices talk to each other?' 'how to build new microservices by leveraging an existing microservice?' or 'how do we compose/integrate microservices and form services/solutions?'

In fact, MSA fosters to build a microservice with limited and a focused business scope. Therefore, when it comes to building IT solutions on top of MSA, it is inevitable that we have to use existing microservices. The interaction between microservices can be done in a conventional point-to-point style; however, that approach becomes quite brittle (with too many point-to-point interactions, which is hard to manage, maintain, scale, and troubleshoot) when it comes to microservices solutions with several services. Therefore, we need to adhere to the best practices of integrating microservices that eliminate the drawbacks of point-to-point style interactions.

- Using a gateway to expose microservices: Use a gateway to front all your microservices and all consumers use the microservices through the gateway only.
- No direct calls among microservices: Microservices cannot invoke other microservices directly; all calls must go through the gateway.
- Micro-integration via an integration server.

Now let's have a look at the techniques related to the interaction between microservices.

14.1 Orchestration at the Microservices Layer

When you have to call multiple microservices to support a given business requirement you can build another microservice (which again addresses a limited business scope) that will orchestrate the service calls to the required microservices and aggregate the final response and send that back to the original consumer.

For example, Figure 14 depicts a scenario in which we have a few microservices namely A, B, C, and D. Now we want to introduce a new business functionality that requires to call microservices A and C sequentially and provide an aggregated response. For this, we can build a new microservice (microservice E) and the orchestration logic that contains calling service A and C is embedded into microservice E. All invocation of microservices are done through the gateway. If microservice E has to be scaled independently, that can be done by scaling microservice E, A, and C as required.

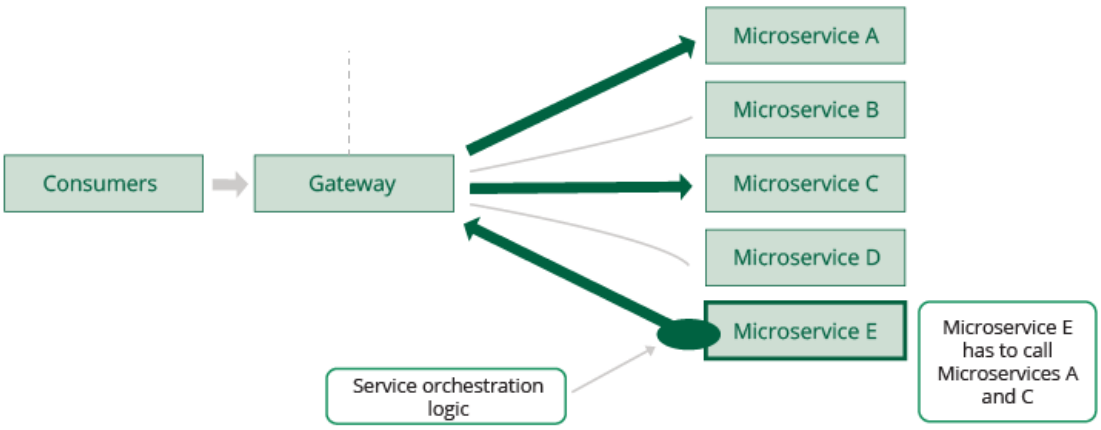

Figure 14: Service orchestration implemented at the microservices level

14.2 Orchestration at the Gateway Layer

The other possible approach is to implement the same orchestration scenario by bringing in the orchestration logic to the gateway level. In this case, we don't have to introduce another new microservice, but a virtual service layer hosted in the gateway will take care of the orchestration.

For example, as shown in Figure 15, the service calls to microservices A and C can be implemented inside the

gateway layer (most microservice gateway implementations support this feature).

When it comes to scaling the newly introduced business functionality, we have to scale the gateway, and microservices A and C. With this, the gateway will become somewhat monolithic because it's also responsible to route all other microservices requests.

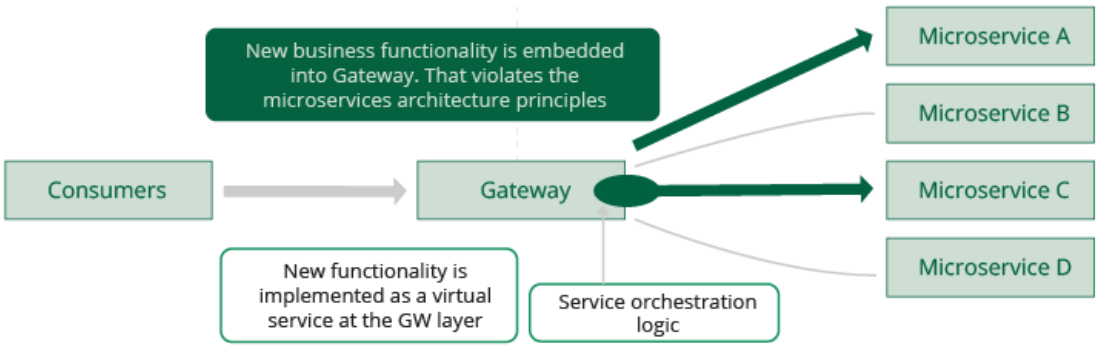

Figure 15: Service orchestration implemented at gateway level

14.3 Micro-Integration

When we have to build integration solutions, often it is an overhead to use a centralized server that contains the integration logic. The concept of micro-integration envisions a lightweight integration framework that can be used to build integration solutions; it can integrate microservers and/or other services/systems (on-premise or SaaS). We only run that integration scenario per each runtime of the integration engine. This must be a runtime that's extremely lightweight (starts within a couple of seconds, and has a low memory footprint). We can then scale this runtime as required. This is a major difference from the conventional central integration server approach where you can't scale only a selected integration scenario, but rather you have to scale the monolithic runtime along with all the deployed integration scenarios.

14.4 Choreography Style

Another possible approach is to build interactions among microservices using asynchronous messaging style, such as MQTT or Kafka. In this case, there is no central component that will take care of service interactions. Various services can do pub-subbased messaging using messaging protocols.

15. WSO2 Microservices Framework for Java (WSO2 MSF4J)

There are quite a few libraries and frameworks to build microservices, but most of them don't really adhere to the core principles of microservices, such as being lightweight or container friendly.

WSO2 offers a microservice framework that's lightweight, fast, and is container friendly.

WSO2 Microservices Framework for Java (WSO2 MSF4J) offers the best option to create microservices in Java with container-based deployment in mind. Microservices developed using WSO2 MSF4J can boot in just a few milliseconds in a Docker container and can easily be added to a Docker image definition.

16. 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
- Most enterprises 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 - that means you need to have integration software, such as an ESB, to cater to all your enterprise integration needs
- All business functions should be exposed as APIs by leveraging API management techniques
- Interaction between microservices should be supported via a gateway
- Service orchestration between microservices may be required for some business use cases and that could be implemented inside another microservice or gateway layer that can do the orchestration