API-driven Microservice Architecture

A WSO2 Reference Architecture

Version Q1-2019

Author

- Dakshitha Ratnayake, Enterprise Architect - CTO Office dakshitha@wso2.com

> *Microservice Architecture (MSA) is an excellent approach to building decentralized systems. However, microservices are too granular when it comes to architecting larger systems and projects in the brownfield. Most enterprises follow a layered architecture with both Service-oriented Architecture (SOA) principles and MSA concepts by grouping the services or microservices into layers in the overall enterprise architecture. This approach makes each architecture layer a logically centralized set of shared components.*

> *This paper will introduce microservices and will predominantly discuss the layered approach for an API-driven MSA. It will introduce ways of gradually transitioning from a monolithic architecture to a layered MSA via the API gateway pattern using WSO2 middleware and recommended technologies. This paper will also briefly cover other reference architectures such as segmented architecture, which is a subpattern of layered architecture, and the alternative reference architecture known as cell-based architecture. This paper will only cover the request-response communication style for client-microservice communication and a separate paper will discuss the event-driven communication style.*

1.0 Introduction

In an age when delivering great digital experiences is more important than ever, business success lies in offering agile digital services with high customer satisfaction. There needs to be an alignment between the overall corporate strategy and the pursued digital initiatives in order to transform the core business architecture to a digital architecture. A digital architecture is one that fosters rapid integration of new technologies to fuel digital transformation. To elaborate, a digital architecture is composed of a stack of layers that support the business value chain. Underpinning it all is the technology layer, which encompasses the application, API management, security, analytics, integration, services and data layers, and core infrastructure.

digital_architecture

Figure 1 - A Typical Digital Architecture

A microservice architecture (MSA) is a method of developing software applications by building them as collections of independent and modular services. Microservices can be deployed to create a more

flexible digital architecture. Smaller teams can work separately on these microservices and share them with other application developers via APIs for reuse. In fact, these individual services can be released, scaled, and updated without impact to the rest of the system. When coupled with a containerized environment, MSA is the ideal option for companies that must deliver and scale fast and enable support for a range of platforms and devices. When adopting an MSA, its constituent microservices can be categorized into several types and subtypes.

At the same time, most organizations are not fully ready to adopt a pure MSA. Even though an MSA approach is followed for newer development, many conventional systems, such as SOA systems, Enterprise Service Bus (ESB), PaaS/SaaS applications, and any other monolithic vertical deployments of legacy applications will remain a part of an enterprise's overall topology. These brownfield systems cannot be ignored and will have to be accessed by the microservices (at least during the transition period). These systems and services will have to be exposed via managed APIs for seamless internal and external access.

By considering the above facts, the holistic view of an MSA-enabled digital architecture can be divided into three zones as given below:

- Zone 1: Inner Architecture
- Zone 2: Outer Architecture
- Zone 3: External Architecture

Inner Architecture	Core Microservices - Microservices which purely contain business logic and data services.
Outer Architecture	Utility Microservices - Includes integration microservices, micro-gateways, security token services, micro-brokers or any other micro-runtime.
External Architecture	End-user applications, legacy systems and data, SaaS applications, partner systems, etc.

Table 1 - MSA Architecture Zones

Consequently, an MSA-enabled digital architecture will closely resemble the architecture depicted in Figure 2 below.

msa_enabled-da

Figure 2 - An MSA-enabled Digital Architecture

For this purpose, this paper will explain how to achieve an MSA-enabled digital architecture. It will also highlight the best practices of adopting an API-driven microservice architecture in order to build a digital architecture iteratively, whether it is completely greenfield or brownfield, and how the WSO2 product stack and its derivatives can help achieve these MSA goals.

2.0 Microservice Architecture (MSA)

2.1 Key Characteristics

MSA is an application architecture pattern in which a large application is broken down into many loosely-coupled microservices. The goal of microservices is to sufficiently decompose the application in order to facilitate agile application development and deployment.

In contrast to a more classic monolithic application, in which everything is tightly-coupled and deployed as one big chunk, a microservice architecture tries to decouple all modules, where each service has its own unique and well-defined scope and runs in its own process. Each microservice should ideally own a single responsibility. By enabling small autonomous teams to develop, deploy and scale their respective services independently, microservices enable parallel development, thereby speeding up the production cycle significantly.

MSA is an evolution of Service Oriented Architecture (SOA) and adopting an MSA is closely correlated with the use of DevOps and continuous integration and continuous delivery (CI/CD). Classic SOA is often implemented inside deployment monoliths and is more platform driven, while microservices must be independently deployable and therefore offer more flexibility in all dimensions. Cloud native microservices (microservices that exploit the advantages of cloud computing, container packaging, and dynamic management) take the SOA concept to a new level where the cloud infrastructure enables services to be implemented and managed at scale. With Docker and Kubernetes providing effective ways to develop, deploy and manage microservices, developers can spin up countless services all at once, monitor the services and provide each service with the resources it needs. Each microservice can be deployed, upgraded, scaled and restarted independent of all the sibling services in the application.

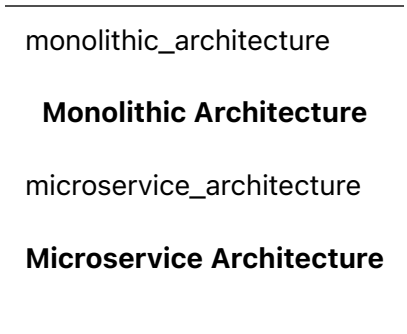

Figure 3 - Monolithic Architecture vs Microservice Architecture

2.2 Microservice Architecture: Benefits and Drawbacks

Benefits

- Individual services are much faster to develop, test and deploy, and much easier to understand and maintain. They are reusable too.
- Each service is developed independently by a team that is focused on that service and are free to choose whatever technologies that make sense. Developers rarely need to coordinate the deployment of changes that are local to their service. These kinds of changes can be

deployed as soon as they have been tested.
- MSA enables the continuous delivery and deployment of large, complex applications.
- MSA enables each service to be scaled independently. Developers can deploy just the number of instances of each service that satisfy its capacity and availability constraints. Moreover, they can use optimized deployment infrastructure that best matches a service's resource requirements.
- MSA creates improved fault isolation. For example, if there is a memory leak in one service, then only that service will be affected. The other services will continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system.

Drawbacks

Like every other technology, the Microservice Architecture has drawbacks and is not a silver bullet. Here is a list of some drawbacks.

- Decomposing data and processing them into independent microservices can increase the complexity of the design.
    - A client application has to know several endpoints to perform any meaningful function and each microservice has to know about at least one or two other microservices e.g. how to interact, IP addresses, etc.
    - Developers must differentiate between internal and external microservices and implement the inter-service communication mechanism between these types. Also, client-side and server-side service discovery is crucial.
    - Developers must also write code to handle partial failure since the destination of a request might be slow or unavailable. They also need to handle distributed transactions to implement use cases that span multiple services.
- Client applications will struggle to consume the services because of variations in interfaces and security protocols due to different teams implementing the microservices.
- Microservices leads to more network calls and hence a monolith given the same hardware usually performs better.
- Developer tools/IDEs are oriented on building monolithic applications and don't provide explicit support for developing distributed applications.

2.3 MSA Implementation

Despite the complexity of microservices, they are usually a good choice for consumer-driven rapid application development and they help to define clear functionality boundaries and self-organizing teams. They create sustainable development, thus promoting agility. This means collaborative development, frequent shipping and frequent engagement with the customer. Each functional area of an application is implemented by its own microservice. Developers can start off by decomposing the application into microservices by business capability, use case or verb, e.g. shipping, or by nouns or resources, e.g. orders. Ideally, each service should have a single responsibility or only a small set of responsibilities.

For inter-process communication, microservices can use synchronous request/response-based communication mechanisms such as HTTP-based REST, Thrift or gRPC. Alternatively, an asynchronous, message-based communication mechanism such as AMQP, MQTT or STOMP can be used. A message format such as JSON or XML, which are human-readable, or a binary format such as Avro or Protocol Buffers can be selected. In order to ensure loose coupling, each service must have its own database rather than sharing a single database with other services.

Microservices can be implemented in any programming language that supports REST and RPC-based services. This paper will focus on the Ballerina programming language in particular. Ballerina is a compiled, transactional, statically and strongly typed programming language with textual and graphical syntaxes that can be used to implement microservices. Ballerina incorporates fundamental concepts of distributed system integration into the language and offers a type safe, concurrent environment to implement microservices with distributed transactions, reliable messaging, stream processing, and workflows.

Once the microservices are defined and implemented, they should be containerized. Docker is the preferred technology for packaging microservices and is a key enabling technology for microservices. Fast immutable deployments, maximizing resource utilization and bare-metal performance are some of the advantages of using Docker. The Docker containers should be dynamically orchestrated using a cloud-agnostic container cluster manager, such as Kubernetes. Developers can independently deploy and scale each microservice, coordinate their deployment through Kubernetes orchestration and collaborate across teams while including tools for managing services and observing the deployments.

2.4 Client-to-Microservice Communication

microservice_architecture

Figure 4 - Microservice Architecture: An Example

Microservices are fine-grained, which means that clients need to interact with multiple microservices. In the retail application in Figure 4, the client application directly consumes several microservices. If the client application wants to list the available products, it will call the products microservice, and if it wants to place an order, it will call the orders microservice. When an order is placed, the orders microservice will have to check the product stock status from the inventory service and talk to the shipping microservice to start the shipping process. In theory, the client could make requests to each of the microservices directly because each microservice has a public endpoint. However, this approach will produce considerable subsequent problems such as the following:

- It introduces the point-to-point spaghetti mess SOA tried to eliminate with the use of an Enterprise Service Bus.
- If there are mismatches between the needs of the client and the fine-grained APIs exposed by each of the microservices, the client will have to make several calls to get what it needs. In a complex application, this could mean hundreds of service calls.

- Consuming services via non-uniform interfaces and protocols One service might use Thrift binary RPC while another service might use the AMQP messaging protocol. Neither protocol is particularly browser- or firewall-friendly and is best used internally. An application should use protocols such as HTTP and WebSocket outside of the firewall.
- Refactoring the microservices will be challenging. Over time, microservices will have to be refactored. For example, change how the system is partitioned into services. If clients communicate directly with the services, then performing this kind of refactoring can be extremely difficult.
- Implementing authentication, authorization, throttling, caching and monitoring for each microservice creates duplication and is unnecessary.

Because of these kinds of problems, it rarely makes sense for clients to talk to microservices directly. When designing and building large or complex microservice-based applications, a good approach to consider is an API Gateway.

2.4.1 API Gateway Pattern

2.4.1.1 API Gateway

Implementing an API Gateway as the single entry point for all clients will address the issues faced in direct client-to-microservice communication.

microservice_architecture

Figure 5 - Microservice Architecture: API Gateway Pattern

In the API gateway pattern, all requests from clients first go through well-defined APIs in a lightweight message gateway (aka API gateway) over REST/HTTP. These APIs route requests to the appropriate microservices. The gateway is mainly responsible for request routing, composition, and protocol translation. The API gateway APIs will often handle a single request by invoking multiple microservices and aggregating the results.

Fundamentally, an API gateway provides for easy developer consumption of microservices as managed APIs and is usually a part of an API management offering. An API management platform should provide ways to onboard and manage developers, create an API catalog and documentation, generate API usage reporting, productize or monetize APIs, and enforce throttling, caching, and other security and reliability precautions. This platform provides the tools, control, and visibility to scale microservices via APIs to new developers and connect them to new systems.

API Gateway: WSO2 Solution Pattern

api_gateway_wso2

Figure 6 - Greenfield MSA: The API Gateway Pattern with WSO2 API Manager

The WSO2 API Manager, recently named as a leader in the Forrester Wave™: API Management

Solutions, Q4 2018 report, is an open source API management solution which provides an API gateway with support for API creation, publication, lifecycle management, versioning, monetization, governance, security, rate limiting and analytics using proven WSO2 technology. It provides web interfaces for development teams to deploy and monitor APIs, and for consumers to subscribe to, discover and consume APIs through a user-friendly storefront. The API Manager consists of several components: API Gateway, Key Manager, Traffic Manager, Publisher Portal, Developer Portal (Store) and Analytics. Depending on the use case and traffic volumes, these components can be deployed in a single runtime or as separate runtimes.

Some microservices projects would be complete re-writes of all the components or just the services layer. In such cases, the developers have the luxury of selecting the technology stack to adopt the MSA pattern. The business logic can be written as Ballerina microservices. The microservices can be directly exposed to client applications via WSO2 API Manager Gateway. The gateway can secure the microservices by using a combination of OAuth2/OIDC at the edge and sending information to the microservices via Java Web Tokens (JWT). The Key Manager can handle the generation and validation of these security tokens. The WSO2 API Manager will also provide other API management capabilities such as documentation, service discovery via the developer portal and analytics for the exposed APIs. WSO2 API Manager can be deployed as the API Gateway as shown in Figure 6 above. The WSO2 API Analytics profile is always deployed as a separate runtime.

2.4.1.2 Microservice Orchestration

Implementing microservices and having them exposed through an API Gateway layer seems to be a fitting choice when it comes to implementing a greenfield MSA. Even though the popularity of microservices is rising day by day, this doesn't mean that monolithic applications are obsolete. There are applications that are still suited for a monolithic architecture and therefore brownfield environments will continue to exist. So, in most greenfield and brownfield scenarios and with evolving business needs, which will require heavy integration of multiple microservices and other legacy systems, the API gateway alone will not suffice. The capabilities of an Enterprise Service Bus in an SOA, such as message routing, service compositions/chaining, protocol, and message format transformations, implementing resiliency patterns and various other Enterprise Integration Patterns (EIPs), are still required in an MSA to create a cohesive experience. How should these capabilities be absorbed into an MSA?

Integration Microservices

In a typical SOA, virtual composite services are exposed from an ESB layer to integrate multiple services and other systems. The ESB is used as the centralized bus that connects all these services and systems. However, this is an anti-pattern in MSA because any system that requires a lot of central management can become problematic as with any monolith. The logic is built into the pipe, thus increasing the coupling in the system. Furthermore, owing to its fundamental incompatibilities, the ESB is unable to fit into an MSA.

As an alternative to a centralized ESB, implementing integration microservices is the best option from an MSA point of view. In this approach, integration or composite services will be hosted as a

second layer of microservices. These services often have to support ESB functionality such as routing, transformations, orchestration, resilience and stability patterns. If we take the retail application example, we can build an integration microservice which consumes the orders and shipping microservices as shown below. As opposed to the monolithic ESB approach, all ESB/integration functionalities are segregated and dispersed among all the integration microservices.

integration_services

Figure 7 - Microservice Architecture: Integration Microservices

WSO2 has come up with two ways of implementing such integration microservices.

For a configuration approach, a modern microservices-friendly ESB runtime known as the WSO2 Micro Integrator can be used. WSO2 Micro Integrator is a lightweight runtime built using the same technology as the WSO2 Enterprise Service Bus. The logic is authored in XML. Various Enterprise Integration Patterns can be implemented and numerous other applications can be connected via connectors.

On the other hand, Ballerina is a programming language and platform that aims to fill the gap between integration products and general purpose programming languages. Ballerina makes it easy to create microservices that integrate and orchestrate across distributed endpoints. It attempts to provide agility and integration simplicity simultaneously and it achieves this by packaging a language, integration syntax, and environmental deployment constructs into a single code file which is compiled into a binary, executable within a virtual machine, and then made part of a developer's toolchain with IntelliSense language servers and debuggers for IDEs.

Integration Microservices: WSO2 Solution Pattern

integration_services_wso2

Figure 8 - Microservice Architecture: Integration Microservices with Ballerina or WSO2 Micro Integrator

As shown in Figure 8, both the business logic and integration logic can be written as microservices and will reside in separate runtimes. Integration services, which integrate the microservices, can be exposed to the client applications via the gateway. Ballerina can be used to write these integration microservices. At the same time, the WSO2 Micro Integrator can also be used to write the integration logic using the configuration approach.

2.4.1.3 Decentralized Gateways

Decentralized gateways are extremely useful when it comes to scaling. For example, assume a scenario when multiple instances of the products microservices are required but only one or a few instances of the orders microservices are needed. This is a realistic scenario where many requests will be received by a particular set of microservices, such as the products microservice (to browse

the product catalog), but not as many requests will be received by the orders microservice (to place orders). Imagine a single monolith API gateway architecture where all the APIs reside on the monolith gateway and microservices behind that API gateway. To scale the products microservice, the monolith gateway will have to be scaled as well. Scaling one part of the system results in scaling the entire gateway deployment. To prevent such a situation, a decentralized gateway is a good option where there is a microgateway per microservice. Whenever a microservice needs to be scaled, only the gateway that fronts this microservice will be scaled along with the microservice.

microgateways

Figure 9 - Microservice Architecture: Decentralized Gateways

Decentralized Gateways: WSO2 Solution Pattern

WSO2 API Microgateway is an immutable, ephemeral and lightweight API gateway. The microgateway is used for message security, transport security, routing, and other common API management services. The WSO2 Microgateway is designed to scale, where it does not require any mandatory connections to other components, such as the Key Manager, Analytics, Traffic Manager, etc. as it is done with the standard monolithic WSO2 API Manager. This way it can be made to scale easily without having to worry about scaling the other components. This is achieved through self-validating tokens, localized rate limiting, offline analytics, and immutability. The microgateway has native support for Docker and Kubernetes, and developers can obtain Docker images and Kubernetes artifacts as the build outcome of a microgateway.

wso2_microgateways

Figure 10 - Microservice Architecture: Decentralized Gateways with WSO2 Microgateways

In Figure 10, the disaggregated gateway approach is followed, where each gateway is an instance of the WSO2 Microgateway and a single independent microgateway can front each microservice or integration service. The gateway APIs must be defined in the WSO2 API Manager with API information, target endpoints, etc. A microgateway toolkit, when executed with certain parameters, will connect to the API manager, download the definitions and policies of the APIs to the toolkit and spin up a microgateway instance. For example, if a microgateway needs to be created for the products microservice, a products API must be first defined and published in the API manager via the Publisher and Store portals respectively. The toolkit will download the definition of the products API and all the policies associated with it and generate the microgateway runtime for the products API. Alternatively, an API definition (Swagger file) can be provided instead of creating the APIs in the API Manager. The size of the runtime, boot up time, memory, disk space can be defined. This runtime can run on a virtual machine or a Docker container.

The microgateway supports the standard OAuth2 flow, where a client application will first get a signed JWT token issued by the Security Token Service (STS), which can be the Key Manager of the WSO2 API Manager or any other STS. When the client application sends the signed JWT to the microgateway, the microgateway can self-validate the token as the STS is trusted by the

microgateway. In this particular scenario, the microgateway has no link to the Key Manager. If the microgateway needs to be scaled, it can be done without having to scale the STS. For analytics, meta information of the API requests is logged into data files residing in the local file system of the microgateway (shared or private). Subsequently, these files will be uploaded periodically to the API Analytics server (this process can run in the gateway process itself or in a separate process). The API Analytics server will analyze the raw data and generate business intelligence. For traffic management, rate limiting policies are embedded into the microgateway runtime.

2.4.1.4 API Gateways vs Service Meshes

One of the frequently asked questions regarding the API gateway is how it is different from a service mesh. A service mesh handles microservice-to-microservice interactions and are used for complicated routing. It can perform tasks such as load balancing, service authentication, service discovery, routing, and policy enforcement. A service mesh can also provide detailed insights into the microservices environment, control traffic, implement security, and enforce policies for all services within the mesh. There are two logical components that create a service mesh: a data plane and a control plane. The sidecar, which is a proxy to the microservice, falls under the data plane. Other than sidecar proxies, all service mesh solutions have a controller (control plane), which defines how sidecar proxies should work. The service mesh control plane is the central place to manage the service proxies and it records a lot of network information.

Istio is the most popular and comprehensive service mesh platform in the market at the time of this writing. By using the sidecar model, Istio runs in a Linux container in Kubernetes pods and injects and extracts functionality and information based on the controller configuration. This configuration lives outside of the microservice code and therefore makes the code small and simple by moving operational aspects away from code development.

service_mesh

Figure 11 - Control Plane and Data Plane in a Service Mesh

Even though the API gateway and the service mesh have overlapping capabilities, the key difference between them is that the API gateway is fundamental in exposing microservices as managed APIs to external (outside the MSA) parties whereas the service mesh is merely an inter-service communication infrastructure which doesn't have any business notion of the entire solution. Service meshes in their native form have an API management gap that requires to be filled. These are related to exposing services to external consumers (advanced security, discovery, governance, etc), business insights, policy enforcement, and monetization.

The service mesh and API gateway are complementary such that they live at different levels and solve different problems. They can exist independently but co-exist complementarily.

api_gateways_and_service_mesh

Figure 12 - Istio Mixer Adapter for WSO2 API Manager

The WSO2 API Manager can be positioned to integrate with Istio and manage microservices deployed in Istio as APIs via the Istio mixer adapter for WSO2 API Manager. The WSO2 API Manager goes beyond and provides broader business capabilities such as designing, publishing, documenting, analyzing and monetizing APIs in a secure environment.

The Istio mixer adapter for WSO2 API Manager can secure services using JWT and OAuth2 tokens, validate subscriptions and scopes, use WSO2 API Manager Analytics for business insights and more.

3.0 Microservice Architecture - Reference Architectures

3.1 Layered Microservice Architecture

Most enterprises follow a layered architecture with both SOA principles and MSA concepts by grouping the services or microservices into layers in the overall enterprise architecture. This approach makes each architecture layer a logically centralized set of shared components.

layered_msa

Figure 13 - Microservices Layered Architecture

In this MSA pattern, functional capabilities are grouped in layers by following a system of systems view. It is a centralized system where data moves from layer to layer. With the gateway layer also being disaggregated, a pure MSA can be implemented with different types of services that can be categorized into a few different layers as shown in Figure 13 above.

Core/Atomic microservices are placed at the bottom layer. They are extremely fine-grained self-contained services (with no external service dependencies) comprising the business logic with little or no network communication logic.

Composite/Integration microservices handle the composition of multiple atomic/core services and the middle layer consists of such composite or integration microservices. These services often have to support a significant portion of ESB functionality such as routing, transformations, orchestration, resiliency and stability patterns. They are more coarse-grained than atomic services and independent from each other. They contain routing logic, data mapping logic and network communication logic which handles inter-service communication through various protocols and resiliency behaviors such as circuit breakers. These services also could bridge the other legacy and proprietary systems (e.g SAP ERP), external web APIs (e.g. Salesforce) and shared databases (often known as the anti-corruption layer).

API/Edge microservices are the third type of microservices and topmost in a layered architecture. Composite services or even some atomic services can be exposed as managed APIs using API/Edge services. These services are a special type of composite services that apply basic routing capabilities, versioning, API security patterns, throttling, monetization and more.

3.1.1 Layered Architecture with WSO2: A Greenfield Deployment

greenfield_layered_msa_wso2

Figure 14 - Microservices Layered Architecture with the WSO2 Technology Stack

Management of microservices is handled by WSO2 API Microgateway, which provides secure, low-latency access to microservices and eliminates the need for a central gateway by enabling enterprises to apply API management policies in a decentralized fashion. Microservices integration is optimized using Ballerina or WSO2 Micro Integrator. In addition to the Key Manager of the API manager, which takes care of token generation and validation, the WSO2 Identity Server can be used for identity federation, single sign-on, identity bridging, adaptive and strong authentication, and account management and identity provisioning as a cross-cutting component of the MSA.

3.1.2 Layered Architecture with WSO2: A Brownfield Deployment

Most microservices projects are implemented in brownfield environments which already have legacy, cloud, and other services, an ESB environment, etc. In such a scenario, a complete rewrite is usually costly. A typical approach is where newer services are implemented as microservices and a separate layered architecture (integration microservices and microgateways along with the core microservices) will be formed while the existing SOA environment (web services, APIs, legacy systems and ESB layer) remain as-is. The ESB will continue to integrate multiple services and other systems and will be used as the centralized bus that connects all these services and systems outside the MSA. These endpoints on the ESB will be exposed to the client applications through the centralized API gateway where they will be secured, throttled and monitored.

brownfield_layered_msa_wso2

Figure 15 - MSA Layered Architecture and SOA in Coexistence with the WSO2 Stack

In the architecture above, a layered MSA also coexists with the SOA. New functionality can be implemented within the MSA and the existing services and applications in the SOA can reside as-is.

3.1.2.1 Transitioning from a Monolith to a Microservice Architecture

An advantage of introducing a gateway to an MSA is the ability to smoothly and continuously move from a monolith app to an MSA. Rewriting a large monolith application from scratch is a massive effort and has a good amount of risk associated with it. This also prevents users from using the new system until it is complete. There will be a lot of uncertainty involved until the new system is developed and functioning as expected. Consequently, there will be minimal enhancements or new features delivered on the current platform, so the business will have to wait to have any new features developed and released. This task will, more often than not, have to be phased out.

Known as the Strangler Pattern, this is a design pattern to incrementally transform the monolith into microservices by replacing a particular functionality with a new service. Once the new functionality is ready, the old component is strangled, the new service is put into use, and finally, the old component is decommissioned altogether. However, when phasing out, chances are that there will be a lot of change. The microservice interfaces will change and doing this over time will result in a

nightmare for consuming applications. To make the client applications agnostic to such frequent changes, the API gateway can be used as the Strangler Facade.

strangler_facade

Figure 16 - Microservice Architecture: The Strangler Facade in the Strangler Pattern

As the Strangler Facade, the API gateway will act as a proxy layer with static or well-defined interfaces corresponding to the backend microservices. The subsequent changes in the microservices will not affect the client apps as they will only deal with the API gateway layer.

3.2 Segmented Microservice Architecture

The segmented architecture is a subpattern of the layered architecture, which is created by dividing the layered architecture into small segments based on the functional capabilities within each architecture layer and organizational ownership. It is also a centralized system where data flow moves from layer to layer. This pattern segregates the layers of each subdivision and makes the subdivision responsible for its overall functionality at each layer.

segmented_msa

Figure 17 - MSA Segmented Architecture

3.2.1 Segmented Microservice Architecture with WSO2

segmented_msa_wso2

Figure 18 - MSA Segmented Architecture with WSO2 Technology Stack

Similar to the layered architecture, the WSO2 components can be used in a layered fashion with cross-cutting identity and access management with WSO2 Identity Server. Each layer will now be segmented to suit business needs and can be implemented separately.

3.3 Cell-based Microservice Architecture

(This section is based on the Cell-based Reference Architecture paper [13]. Prior reading is recommended.)

A segmented architecture is too high-level to enforce a decentralized, self-contained architecture unit. At the same time, microservices are usually too fine-grained to be treated as an architecture unit. In a cell-based architecture, functional capabilities are grouped in an architecture unit, known as a cell, based on scope and ownership. A cell is independently deployable, manageable, and observable. Components inside the cell can communicate with each other using supported transports for intra-cell communication. All external communication must happen through the edge-gateway or proxy, which provides APIs, events, or streams via governed network endpoints using standard network protocols. Teams can self-organize to produce units of architecture which are

continuously deployed and incrementally updated. The cell-based architecture goes beyond the traditional layered architecture and creates a framework for decentralization.

The gateway is the control point for a cell-based architecture, which provides a well-defined interface to a subset of APIs, events, and streams. In this pattern, the gateway becomes the only access point (endpoint) for the cell. As a result, the gateway acts as a policy enforcement point, an observability touchpoint, and an enabler for governance frameworks. The cell-based architecture can work on a local security model within the cell or extend to a federated security model (which is common) by connecting beyond the boundary of the cell.

cell	**cell_wso2**
A Single Cell	***Cell: A WSO2 Mapping***

Figure 19 - Cell: A Self-contained Architecture Unit

Figure 20 depicts a portion of a cell-based architecture mapped with WSO2 components where application functionality is divided into multiple cells. Each cell contains different components for building the expected functionality.

cell_implementation_wso2

Figure 20 - Cell-Based Architecture: A WSO2 Implementation

4.0 Installation Experience

4.1 Continuous Integration and Continuous Deployment

To create great digital experiences, adopting the ‘continuouses’ is no longer a ‘nice-to-have’ because speed to market can break a company or enable it to survive and thrive into the future. In many leading businesses, the shift from a single development pipeline to multiple pipelines, and from waterfall development to automated continuous integration/continuous delivery (CI/CD) processes that facilitate rapid iteration, is already happening, and microservices are a major driver. The distributed and independent nature of microservices lends itself naturally to implement continuous integration (CI) and continuous deployment (CD). The infrastructure required for the team to integrate, test, and deploy any changes must be completely automated. This promotes a highly responsive user experience.

Each microservice will have its own CI/CD pipeline. CI/CD channels need to be set up so that changes to the source code automatically result in a new container being built, tested and deployed in staging and eventually pushed to production. Autonomous teams will own the entire lifecycle of a microservice. The team will continuously work on enhancing the features of the microservice and fixing the issues that are encountered in production. The CI/CD pipeline for each microservice will be as shown in Figure 21.

ci_cd

Figure 21 - CI/CD Pipeline for Microservices

ci_cd_msa

Figure 22 - CI/CD Pipeline for Microservices

1. The team commits the changes to a version control repository such as Git.
2. A CI/CD system such as Jenkins can be used to build the changes and run the unit tests for the microservice for which the change is committed.
3. If there are any test failures, an alert is sent back to the team. This process repeats until all the tests succeed.
4. Once the tests succeed, the CI/CD system merges the changes with the mainline and prepares a release artifact for the service. Artifacts for microservices are packaged as Docker containers that are readily deployable by an orchestration tool such as Kubernetes.
5. Thereafter, the CI/CD system publishes the release to a central repository and instructs the orchestration tool to pick the latest version of the microservice that contains the recent changes.
6. The orchestration engine then pulls the latest release from the repository and deploys it in production.
7. All the instances in production are monitored by automated tools that generate alerts for the team if there are any issues. Once alerted, the team fixes the issues and submits the change request to the version control system that triggers the build. The entire process then repeats to push the changes to production.

The orchestration engine generally does a rolling upgrade of the service while deploying updates. This enables the microservices to be upgraded without any downtime. Some teams prefer to do a Blue/Green deployment while others prefer a Canary Release.

The core microservices and integration microservice written in Ballerina in an MSA can seamlessly fit into a CI/CD pipeline. All the changes in a microservice can be committed to a version control repository, such as Git. Ballerina is shipped with its own build system and module management and also works with CI/CD tools such as Jenkins and Codefresh. It can generate deployment artifacts during compilation and run unit tests. Furthermore, it has built-in container support; in other words, Ballerina provides annotations to generate Docker images and Kubernetes artifacts.

To automate the deployment of micro gateways and their APIs, the REST APIs of the WSO2 API Manager Publisher can be used in various ways in the CI/CD process implementation. Periodic or manually triggered automation jobs can read API metadata from one environment, create or update them in the next environment, and ideally run API tests on the higher level environment that make sure no contract is broken by the changes introduced. The API tests can be done via JMeter or SOAP UI test suites that make API calls in the microgateway to verify responses received. Configuration management tools such as Chef, Ansible, and Puppet can be used to automate configuring a microgateway/API manager instance. Since the microgateway has native support for

Docker and Kubernetes, Docker images and Kubernetes artifacts can be obtained as the build outcome of the configured microgateway.

4.2 The Technology Stack

Organizations that do implement microservices often adopt the MSA style along with most of the complementary technologies listed below.

WSO2 Technology	Ballerina Microgateway Micro Integrator API Manager Enterprise Integrator Identity Server
IDE	Eclipse IntelliJ VSCode
Code Repository	Github Gitlab
Binary Repository	Maven Ballerina Central JFrog
CI/CD	Jenkins Travis Codefresh
Test Frameworks	SOAP UI JMeter TestNG
Configuration Management Tools	Puppet Chef Ansible
Container Orchestration	Kubernetes OpenShift Mesosphere DC/OS Hashicorp Nomad Docker Swarm

Service Mesh	Istio Linkerd Conduit nginMesh

Observability	Monitoring
	• Application Metrics
	  ○ WSO2 Analytics Server (WSO2 API Analytics, WSO2 Integrator Analytics, etc.)
	  ○ Prometheus/Grafana
	  ○ Splunk
	  ○ AppDynamics
	• System Metrics
	  ○ Icinga
	  ○ AWS Cloud Watch
	
	Logging
	• Fluentd
	• ELK Stack
	  ○ Beats - Agents that ship data to Logstash or Elasticsearch. Filebeat will ship the WSO2 logs to Logstash.
	  ○ Logstash - Processes and structures the log files received from Filebeat and sends to Elasticsearch.
	  ○ Elasticsearch - Stores and indexes the logs received by Logstash.
	  ○ Kibana - Visualizes the data stored in Elasticsearch
	
	Distributed Tracing
	• Jaeger
	• Zipkin
	• AppDash
	• AppDynamics
	• WSO2 Analytics Server (WSO2 API Analytics, WSO2 Integrator Analytics, etc.)

Table 2: Useful Technologies for an MSA

5.0 Summary

Microservice Architecture is the ideal choice for building complex and evolving applications. They

create sustainable development, thus promoting agility and scalability. MSA is also not for everyone - its prerequisites and disruptive impact must be understood before determining where, when and whether to use it. Also, older monolithic architectures will remain part of an enterprise's overall topology mainly because it is costly to eliminate them completely. One way to manage the speed and flexibility that microservices provide while tackling complexity is by using APIs. A fitting API strategy makes microservices easier to manage and allows them to coexist with existing legacy systems. Therefore, combining an MSA with a holistic API strategy is a proven way of reaping the benefits of microservices while limiting the drawbacks.

This is why implementing an API-driven MSA is critical to making microservices effective for most businesses. APIs are not just low-level programming code interfaces anymore. They are now widely productized, adhering to standards like REST, which make them developer-friendly with management and governance. The API gateway pattern for microservices addresses all these aspects with API management, which is a capability to provide control, visibility, and governance over these increasingly valuable business assets.

While Ballerina simplifies the development of microservices and helps achieve integration simplicity with its first-class integration syntax, WSO2 provides a compelling platform for API management, integration and identity and access management to facilitate an API-driven MSA. By delivering functionality on a cloud native, open source platform, WSO2 facilitates agility by extending platform-wide support for the development and deployment of distributed, lightweight microservices.

References

[1] Ballerina https://ballerina.io/

[2] WSO2 API Manager https://wso2.com/api-management/

[3] WSO2 Enterprise Integrator https://wso2.com/integration/

[4] WSO2 Identity Server https://wso2.com/identity-and-access-management/

[5] Microservices https://thenewstack.io/category/microservices/

[6] Michelle Gienow - Microservices 101]https://thenewstack.io/microservices-101/

[7] Chris Richardson - Pattern: Microservice Architecture https://microservices.io/patterns/microservices.html

[8] Samir Behara - Monolith to Microservices Using the Strangler Pattern https://dzone.com/articles/monolith-to-microservices-using-the-strangler-patt

[9] Martin Fowler - Strangler Application https://www.martinfowler.com/bliki/StranglerApplication.html

[10] Istio Service Mesh https://developers.redhat.com/topics/service-mesh/

[11] Kasun Indrasiri - Service Mesh vs API Gateway https://medium.com/microservices-in-practice/service-mesh-vs-api-gateway-a6d814b9bf56

[12] Kasun Indrasiri - Microservices Layered Architecture https://medium.com/microservices-in-practice/microservices-layered-architecture-88a7fc38d3f1

[13] Asanka Abeysinghe and Paul Fremantle - Cell-based Reference Architecture: Version Q2-2018 https://github.com/wso2/reference-architecture/blob/master/reference-architecture-cell-based.md

[14] WSO2 Named a Leader | The Forrester Wave™: API Management Solutions, Q4 2018 https://wso2.com/resources/analyst-reports/the-forrester-wave-api-management-solutions-q4-2018/?utm_source=landiniam&utm_campaign=forrester_wave_apim_2018B7

[15] Anne Thomas and Aashish Gupta - Innovation Insight for Microservices https://www.gartner.com/doc/3579057/innovation-insight-microservices

[16] Inter-process communication for Microservices https://ballerina.io/learn/by-guide/inter-microservice-communication/

[17] Asanka Abeysinghe - Navigating the Ins and Outs of a Microservice Architecture (MSA) https://www.infoq.com/articles/navigating-microservices-architecture