www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 6 Issue 11 November 2017, Page No. 23199-23208
Index Copernicus value (2015): 58.10 DOI: 10.18535/ijecs/v6i11.20

Containerized Microservices architecture

M.V.L.N.Venugopal

Abstract
Microservices are *the* emerging application platform: It Microservices architecture is not a hype is the architecture that will serve as the basis for many applications over the next few years. to shorten time to market of a software product by improving productivity effect through maximizing the automation in all life circle of the product. Promising container technologies, such as Docker, offer great agility in developing and running applications when combined with microservices-style architecture.
Microservices appreciate approaches emerging technologies like DevOps and continuous delivery in terms of software architecture. with MSA style, several important deployment technologies, such as container-based virtualization and container orchestration solutions, have came in to picture. These technologies efficiently exploit cloud platforms, providing a high degree of scalability, availability, and portability for microservices. Despite enough level of performance, there is still a lack of performance engineering approaches explicitly taking into account the particularities of microservices.
In this paper, we argue why new solutions to performance engineering for microservices are needed. Furthermore, we identify open issues and outline possible research directions. This paper thoroughly studies microservices architectural design along with the various advantages and disadvantages of containerized microservices, architectural advantages, guide lines, goals and the latest technologies used .

Key words: Micro services, Service oriented architecture, Gate Way, Docker, Kubernetes

1. Introduction

Large many Enterprise applications in recent times are designed to facilitate many business requirements. Enterprise architects always look to elevate IT agility and scalability by breaking down business functional models into bounded contexts. Microservices architecture addresses these goals by mapping bounded contexts with autonomous micro units of software (i.e., microservices), each focusing on a single business capability and equipped with well-defined interfaces. Microservices looks to be next development of service oriented architectures and their architecture is fine-grained SOA. Microservices architecture promotes to eliminate ESB as the central bus Microservices is an architectural style inspired by service-oriented computing that has recently started gaining popularity.

Micro services have its impact huge on the recent software industry. MSA is known for the benefits of Scalability, flexibility, and portability "Microservices" latest popular buzz-words in the field of software architecture. Microservices are a new trend rising fast from the enterprise world. It is hard to have definite research solutions for architecting microservices.

Monolithic Architecture

In **software** engineering, a **monolithic** application describes a single-tiered **software** application in which the user interface and data access code are combined into a single program from a single platform. A **monolithic** application is self-contained, and independent from other computing applications

In **monolithic** application a software application hundreds of functionalities bundled into a single application. For examples, ERPs, CRMs, and other various software systems are built as a monolith with several hundreds of functionalities. The Design, development, maintenance and upgrading of such software applications becomes very difficult.

Accounts
Products
Promotions
Orders

Monolithic architecture-Characteristics

- Designed, developed, and deployed as a single unit.
- Difficult to maintain, and enhance.
- Difficult to adopt new technologies and frameworks like Agile.
- Required to redeploy the entire application, to update a part of it.
- scaled as a single application and hard to scale with conflicting resources
- Single unstable service can bring down the whole application.

Consider a retail software application with various services as very good example of a monolithic architecture. All these services are deployed into the same application runtime.

Service Oriented Architecture:

A **service-oriented architecture** (**SOA**) is a style of software design where services are provided to the other components by application components, through a communication protocol over a network. The basic principles of **service-oriented architecture** are independent of vendors, products and technologies. Service Oriented Architecture (SOA) was designed to overcome some of these limitations. SOA application is designed as combination of 'coarse-grained' services, and with broad scope. As per principles of SOA, Services in SOA are independent from each other, and are deployed in the same runtime along with all other services. The very basic goal of SOA is the integration of different software assets, (different organizations), in order to orchestrate business processes.

Micro services Architecture

The foundation of microservices architecture (MSA) is in developing a single application as a suite of small and independent services that are running in its own process, developed and deployed independently. In most of the definitions of microservices architecture, architecture is defined as the process of segregating the services available in the monolith into a set of independent services. microservices is a form of fine-grained service oriented architecture (SOA) implementation that could be used to build independent, flexible, scalable, and reusable software applications in a decoupled manner. . Microservices, pave way for DevOps and Continuous Deployment pipelines.

The microservices architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."- Fowler

The key idea is that by looking at the functionalities offered from the monolith, we can identify the required business capabilities. 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 and each service is addressing a very specific and limited business scope.

Aim of micro services is to improving the development, delivery, and deployment of the software itself. In Micro services every service is required to be an autonomous, independently deployable unit of manageable size and interacts with other services interfaces such as Restful Web APIs. Micro services are enablers of Continuous Delivery and DevOps technologies. Recent technologies container-based virtualization and infrastructure technologies such as Docker and Kubernetes are also utilized to meet challenges like performance testing, Monitoring and Modeling of rapid delivery Due to their highly distributed nature. These facilitate operations by such as rolling updates, automated scaling, and rebalancing in case of node failure. As a consequence, micro service based deployments are much more dynamic and volatile than traditional applications, creating challenges for both monitoring and performance modeling

The same online retail system scenario that already explained above can be realized with microservices architecture as a suite of microservices based on the business requirements, there is an additional microservice created from the original set of services that are there in the monolith. So, it is quite obvious that using microservices architecture is something beyond the splitting of the services in the monolith.

The Principles Of Microservices

1. Modeled Around Business Domain
2. Culture Of Automation
3. Hide Implementation Details
4. Decentralize All The Things
5. Deploy Independently
6. Consumer First
7. Isolate Failure
8. Highly Observable

Accounts
Products
Promotions
Orders

Eight Key Principles for doing microservices well:

- **Model Around Your Business Domain:** Domain-driven design can help you find stable, reusable boundaries
- **Build a Culture of Automation:** More moving parts means automation is key
- **Hide Implementation Details:** One of the pitfalls that distributed systems can often fall into is tightly coupling their services together
- **Embrace Decentralization:** To achieve autonomy, push power out of the center, organizationally and architecturally
- **Deploy Independently:** Perhaps the most important characteristic microservices need
- **Focus on Consumers First:** As the creator of an API, make your service easy to consume
- **Isolate Failure:** Microservice architecture doesn't automatically make your systems more stable

- **Make Them Highly Observable:** With many moving parts, understanding what is happening in your system can be challenging

Architectural principles of microservices

There are few more architecture styles and design principles need to be considered while designing microservices. They are:

1.1.1 Single Responsibility Principle (Robert C Martin)

Each microservice must be responsible for a specific feature or a functionality or aggregation of cohesive functionality.

1.1.2 Domain Driven Design

Domain driven design is an architectural principle in-line with object oriented approach. It recommends designing systems to reflect the real world domains. It considers the business domain, elements and behaviors and interactions between business domains

Service Oriented Architecture

The Service Oriented Architecture (SOA) is an architecture style, which enforces certain principles and philosophies. Following are the principles of SOA to be adhered while designing microservices for cloud.

Encapsulation

The services must encapsulate the internal implementation details, so that the external system utilizes the services need not worry about the internals. Encapsulation reduces the complexity and enhances the flexibility (adaptability to change) of the system.

Loose Coupling

The changes in one Microsystems should have zero or minimum impact on other services in the eco-system. This principle also suggests having a loosely coupled communication methods between the microservices. As per SOA, ReSTful APIs are more suitable than Java RMI, where the later enforces a technology on other microservices.

Separation of Concern

Develop the microservices based on distinct features with zero overlap with other functions. The main objective is to reduce the interaction between services so that they are highly cohesive and loosely coupled. If we separate the functionality across wrong boundaries will lead tight coupling and increased complexity between services.

Guidelines for Designing Microservices

- Single Responsibility Principle(SRP): Having a limited and a focused business scope for a microservices helps us to meet the agility in development and delivery of services.
- During the designing phase of the microservices, find their boundaries and align them with the business capabilities
- Make sure the microservices design ensures the agile/independent development and deployment of the service.
- focus on the scope of the microservices, but not making the service smaller. The true size required should be size to facilitate a given business capability.
- microservice should have a very few operations/functionalities and simple message format.
- Start with relatively broad service boundaries refactoring to smaller ones (based on business requirements) as time goes on.

Goals for Designing Microservices

Microservices are essentially implementation components that communicate with each other using network protocols. Understanding what they are is helpful, but, to some extent, it also misses the point. More important is what microservices enable us to do.

Goals with microservices:

1. Independent deployment of components
2. Independent scaling of components
3. Independent implementation stacks for each component
4. Easy self-serve deployments of components
5. Repeatable deployments of components (external configuration management)
6. Deployments without service interruptions
7. Protection of system availability from individual Instance failure
8. Automatic replacement of component instances when they fail (self-healing)
9. Easy scaling of components by adjusting a simple parameter value
10. Canary testing
11. "Red/black" or "blue/green" deployments
12. Instant reversal of new revision deployments

five architectural constraints (principles that drive desired properties) for the Microservices architectural style. To be a Microservices, a service must be:

1. Elastic
2. Resilient
3. Compassable
4. Minimal, and;
5. Complete

Handling Messages in Microservices

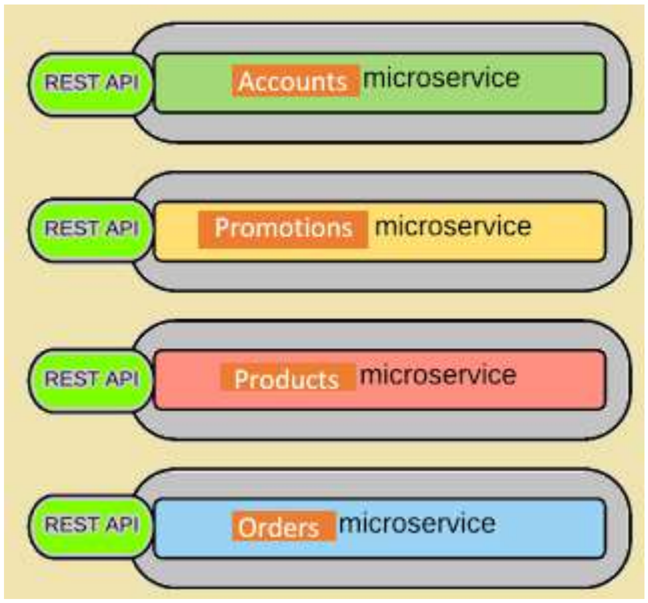
Microservices architecture, have a simple and lightweight messaging mechanism.

There are two types of messages

1. Synchronous Messages
2. Asynchronous Messages

Synchronous Messages –

In synchronous messages, client expects a timely response from the service and waits till it get it. In Microservices Architecture, ReST is the ultimate choice as it provides a simple messaging style implemented with HTTP request-response, based on resource API style. Therefore, most microservices implementations are using HTTP along with resource API based styles i.e. every functionality is represented with a resource and operations carried out on top of those resources. As an alternative to REST/HTTP synchronous messaging, The Apache Thrift can be used for interface definition for microservices and scalable cross-language services development)



Asynchronous Messaging - AMQP, STOMP, MQTT
asynchronous microservices messaging techniques for which client doesn't expect a response immediately, or does not accept a response at all protocols such as AMQP, STOMP are widely used. Another possible approach is to build interactions among microservices using asynchronous messaging style, such as MQTT or Kafka.

Message Formats - JSON, XML, Thrift, ProtoBuf, Avro
In most microservices-based applications, use simple text-based message formats such as JSON and XML on top of HTTP resource API style. In case of binary message formats microservices can leverage binary message formats such as binary Thrift, ProtoBuf (Google's data interchange format) or Avro (Apache Avro™ is a data serialization system.).

Service Registry

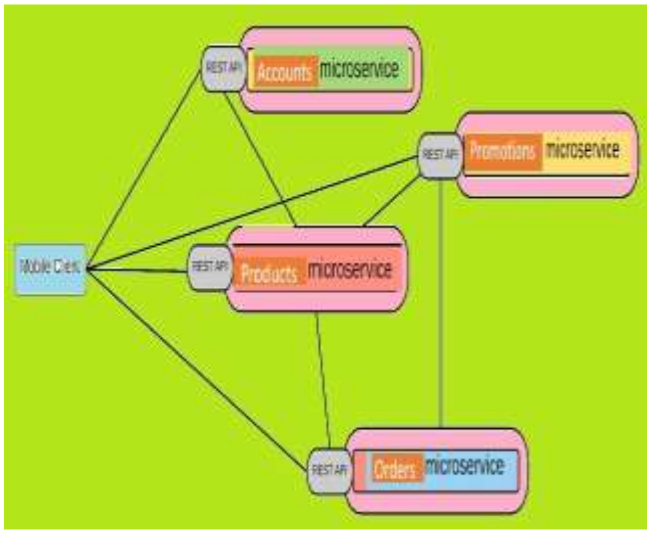
Service Registry holds the microservices instances and their locations. Microservices instances are registered with the service registry on startup and deregistered on shutdown. The consumers can find the available microservices and their locations through service registry.

Service Contracts
As microservices are built on top of REST architectural microservices use the standard REST API definition languages such as Swagger and RAML to define the service contracts. For other microservices implementation which are not based on HTTP/REST (such as Thrift), the protocol level 'Interface Definition Languages (IDL can be used e.g.: Thrift interface description language (Thrift IDL)

Inter-service/process Communication in Microservices
As software applications are built as a suite of independent services, in order to realize a business use case, it is required to have inter-process communication between different microservices/processes. Microservices architecture promotes to eliminate ESB and move the 'smart-ness' or business logic to the services and client (known as 'Smart Endpoints'). Since microservices use standard protocols such as HTTP, JSON, etc. the requirement of integrating with a disparate protocol is minimal when it comes to the communication among microservices. Another alternative approach in Microservices communication is to use a lightweight message bus or gateway with minimal routing capabilities and just acting as a 'dumb pipe' with no business logic implemented on gateway. Based on these styles there are several communication patterns that have emerged in microservices architecture.

1. **Point-to-point**
2. **API-Gateway**

Point-to-point Style
In point to point style, the entirety of the message routing logic resides on each endpoint and the services can communicate directly. Each microservices exposes a REST APIs and a given microservices or an external client can invoke another microservices through its REST API. **Point-to-point Style** model works for relatively simple microservices-based applications but as the number of services increases, this will become very complex.



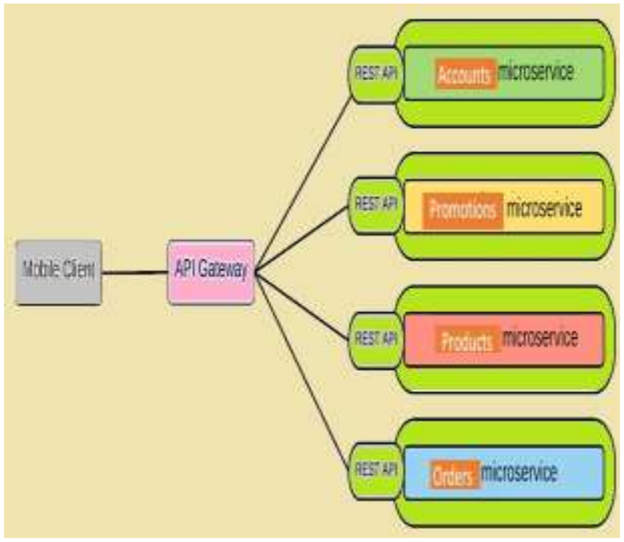
Disadvantages with point-to-pointcommunication.

- The non-functional requirements such as end-user authentication, throttling, monitoring, etc. have to be implemented at each and every microservices level.
- As a result of duplicating common functionalities, each microservices implementation can become complex.
- There is no control at all of the communication between the services and clients (even for monitoring, tracing, or filtering)
- Often the direct communication style is considered as a microservices anti-pattern for large scale microservices implementations.

API-Gateway Style
For complex Microservices, a lightweight central messaging bus which can provide an abstraction layer for the Microservices can be used to implement various non-functional capabilities and the style is termed as API Gateway style. The

Objective behind the API Gateway style is, using a lightweight message gateway as the main entry point for all the clients/consumers and implements the common non-functional requirements at the Gateway level. In general, an API Gateway allows to consume a managed API over REST/HTTP. Therefore, business functionalities can be exposed to be implemented as microservices, through the API-GW, as managed APIs. In fact, this is a combination of Microservices architecture and API-Management which give you the best of both worlds. The API-GW style could well be the most widely used pattern in most microservice implementations.

In our retail business scenario, as depicted in figure 5, all the microservices are exposed through an API-GW and that is the single entry point for all the clients. If a microservices wants to consume another microservices that also needs to be done through the API-GW.

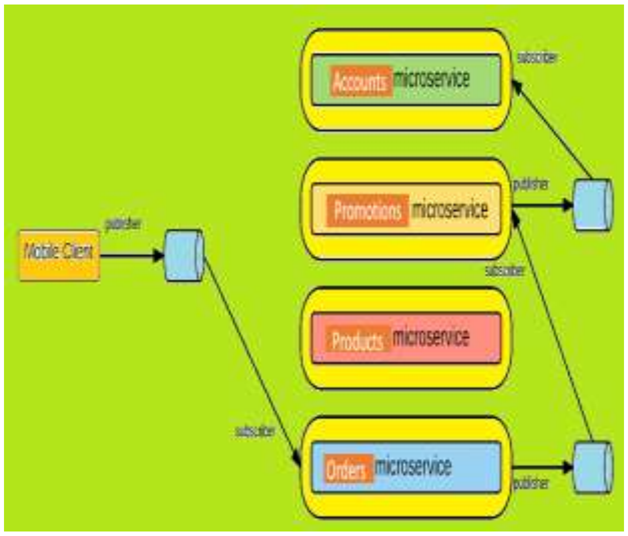


Advantages of Gateway (API-GW) of communication

- Ability to provide the required abstractions at the gateway level for the existing microservices. For example, rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client.
- Lightweight message routing/transformations at gateway level.
- Central place to apply non-functional capabilities such as security, monitoring and throttling.
- With the use of API-GW pattern, the microservices becomes even more lightweight as all the non-functional requirements are implemented at the Gateway level.

Message Broker style

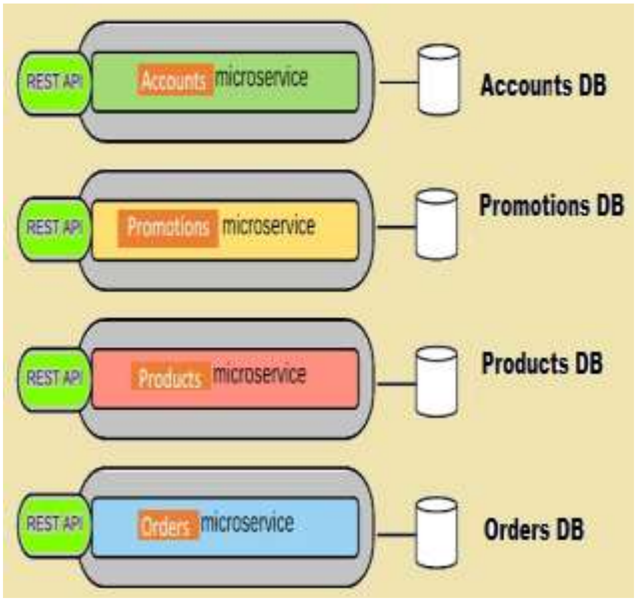
The microservices can be integrated with an asynchronous messaging scenario such as one-way requests and publish-subscribe messaging using queues or topics. A given microservices can be the message producer and it can asynchronously send messages to a queue or topic. Then the consuming microservices 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.



The communication between the consumers/producers is facilitated through a message broker which is based on asynchronous messaging standards such as AMQP, MQTT, etc.

Decentralized Data Management

In monolithic architecture, the application stores data in single and centralized databases to implement various functionalities/capabilities of the application. microservices and, if we use the same centralized database, then the microservices will no longer be independent from each other (for instance, if the database schema has changed from a given microservices, that will break several other services). Therefore, each microservices has to have its own database.



A fie important aspects of implementing decentralized data management

- Each microservice can have a private database to persist the data that requires implementing the business functionality offered from it.
- A given microservices can only access the dedicated private database but not the databases of other microservices.
- In some business scenarios, you might have to update several database 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

The de-centralized data management gives you the 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.

Decentralized Governance

Microservices architecture favors decentralized governance. In microservices architecture, the microservices are built as fully independent and decoupled services with the variety of technologies and platforms.

Decentralized governance capabilities of Microservices

- In microservices architecture, there is no requirement to have centralized design-time governance.
- Microservices can make their own decisions about its design and implementation.
- Microservices architecture fosters the sharing of common/reusable services.
- Some of the run-time governances aspects such as SLAs, throttling, monitoring, common security requirements and service discovery may be implemented at API-GW level.

Service Registry and Service Discovery

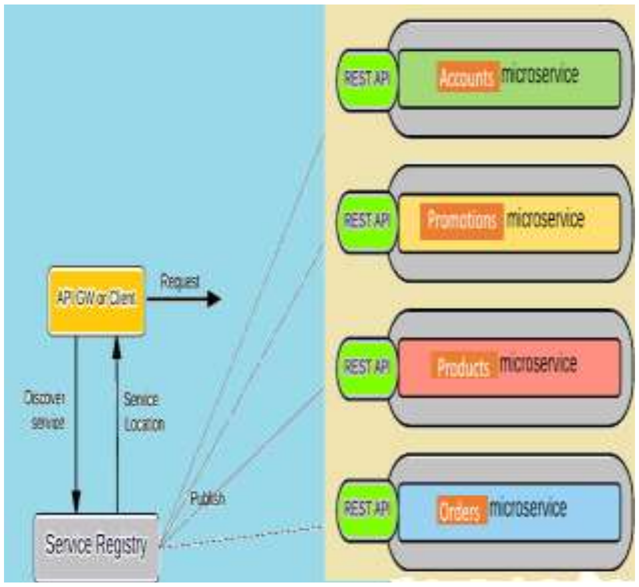
In Microservices architecture, the number of Microservices is quite high. And also, their locations change dynamically owing to the rapid and agile development/deployment nature of microservices. Therefore, it is required to find the location of microservices during the runtime. The solution to this problem is to use a Service Registry.

Service Discovery

To find the available microservices and their location, There are two types of service discovery mechanisms,

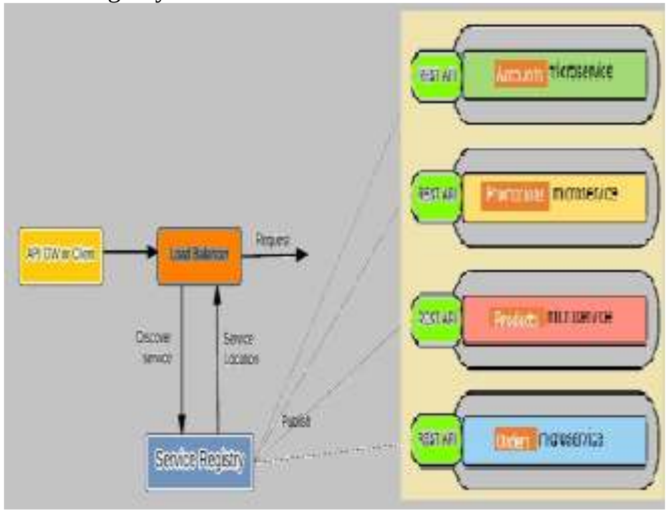
1. Client-side Discovery
2. Server-side Discovery.

Client-side Discovery — In this approach, the client or the API-GW obtains the location of a service instance by querying a Service Registry.



Here the client/API-GW has to implement the service discovery logic by calling the Service-Registry component.

Server-side Discovery — With this approach, clients/API-GW 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 absolute location of the



Deployment Of Microservices

The deployment of microservices plays a critical role and has the following key requirements:

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

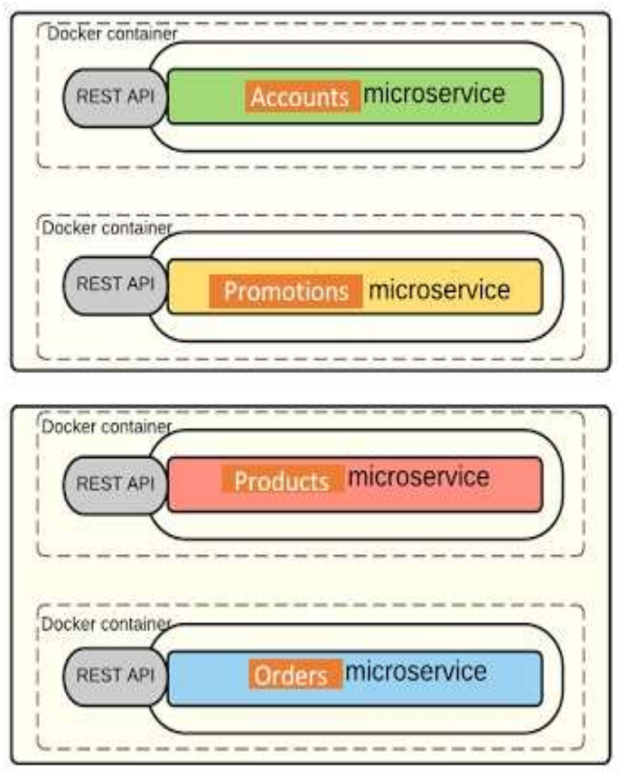
Docker (an open source engine)provides a great way to deploy microservices addressing the above requirements.

The key steps involved are as follows:

- Package the microservices 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 microservices will be much faster as we are using Docker containers (which is much faster than a regular VM)

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. It extends capabilities of Docker Hence using Kubernetes (on top of Docker) for microservices deployment has become an extremely powerful approach, especially for large scale microservices deployments.

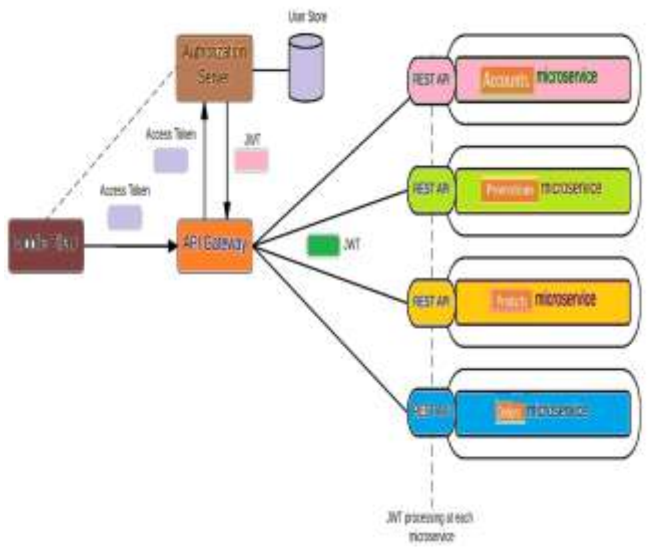


In figure 11, it shows an overview of the deployment of the microservices of the retail application. Each microservice instance is deployed as a container and there are two containers per each host. You can arbitrarily change the number of containers that you run on a given host.

Security

In real world scenarios Securing microservices is very imp[requirement
we can leverage the widely used API-Security standards such as OAuth2 and OpenID
.

- OAuth2 - Is an access delegation protocol.
- OpenID Connect behaves similarly to OAuth, but, in addition to the Access token, the authorization server issues an ID token which contains information about the user.



As shown in figure 12, these are the
Implementing microservices security
key steps involved in implementing microservices security:

- Leave authentication to OAuth and the OpenID Connect server(Authorization Server), so that microservices successfully provide access given someone has the right to use the data.
- Use the API-GW style, in which there is a single entry point for all the client request.
- Client connects to authorization server and obtains the Access Token (by-reference token). Then send the access token to the API-GW along with the request.
- Token Translation at the Gateway - API-GW extracts the access token and sends it to the authorization server to retrieve the JWT (by value-token).
- Then GW passes this JWT along with the request to the microservices layer.
- JWTs contain 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 which is allowing you to transport identity throughout your system.
- At each microservices layer, we can have a component that processes the JWT, which is a quite trivial implementation.

Microservices - Support for Transactions

The microservices architecture encourages the transaction-less coordination between services. Distributed transactions across multiple microservices are an exceptionally complex task. The idea is that a given service is fully self-contained and based on the single responsibility principle. The need to have distributed transactions across multiple microservices is often a symptom of a design flaw in microservice architecture and usually can be sorted out by refactoring the scopes of microservices. However, if there is a mandatory requirement to have distributed transactions across multiple services, then such scenarios can be realized with the introduction of 'compensating operations' at each microservices level. The key idea is, a given microservices is based on the single responsibility principle and if a given microservices failed to execute a given operation, we can consider that as a failure of that entire microservices. Then all the other (upstream) operations have to be undone by invoking the respective compensating operation of those microservices.

Design of Micro services Using patterns

Microservices architecture introduces a dispersed set of services and, compared to monolithic design, that increases the possibility of having failures at each service level. Microservices can fail due to network issues, unavailability of the underlying resources, etc. Unavailable or unresponsive microservices should not fail microservices-based application down. Thus, micro services should be fault tolerant, be able to recover whenever is possible, and the client has to handle it gracefully.

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 the services.

In addition, Gateway can be used as the central point that we can obtain the status and monitor of each microservices as each microservices is invoked through the Gateway.

Commonly used patterns in Microservices

Circuit Breaker

This pattern is quite useful to avoid unnecessary resource consumption, request delay due to timeouts, and also gives us to chance to monitor the system (based on the active open circuit's states).

Bulkhead

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 allow to stop waiting for a response from the micro service, the time interval

Most of these patterns are applicable at the Gateway level. When the microservices are not available or not responding, at the Gateway level we can decide whether to send the request to the microservices using circuit breakers or timeout pattern. Also, it's quite important to have patterns such as bulkhead implemented at the Gateway level, as it's the single entry point for all the client requests, so a failure in a give service should not affect the invocation of the other microservices. Gateway can be used as the central point that we can be used obtain the status and monitor of each microservices

1.2 Microservices in Modern Enterprise Architecture

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. 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 as ESB is no more.MSA encourages the enterprise to build all of its IT solutions as microservices it need to have is a mix of MSA along with conventional architecture of existing systems.

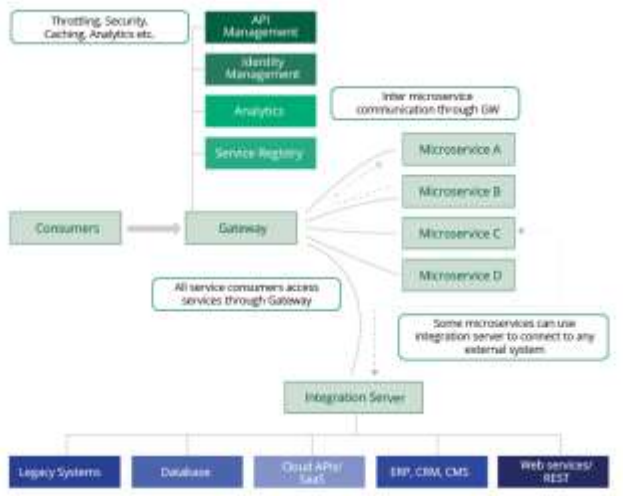

Throttling, Security, Caching, Analytics etc.
API Management
Identity Management
Analytics
Service Registry
Inter microservice communication through GW
Microservice A
Microservice B
Microservice C
Microservice D
Consumers
Gateway
All service consumers access services through Gateway
Some microservices can use integration server to connect to any external system
Integration Server
Legacy Systems
Database
Cloud APIs/ SaaS
ERP, CRM, CMS
Web services/ REST

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.

Why enterprises introduce MSA.

- MSA build solutions to gain the full power that MSA brings in.
- Enterprise integration is still required; As going for a hybrid approach, still need to integrate all internal systems and services with the use ESB.
- the new microservices may need to call monolithic systems to facilitate the various business requirements. In this case, the underlying integration software/ ESB is still useful and the microservices can call the integration server to connect to old systems.
- An organization should look for lightweight, high-performance, and scalable integration software instead of heavyweight integration frameworks.
- API management: Microservices can be exposed via the gateway and all API management techniques can be applied at that layer.
- security, throttling, caching, monetization, and monitoring has to be done at the gateway layer.
- the non-microservices based services (traditional SOA) can also be exposed through the API gateway.

1.3 14. Integrating Microservices

MSA aims to build a microservices with limited and a focused business scope. Therefore, when it comes to building IT solutions on top of MSA, it is inevitable to use existing microservices. The interaction between microservices can be done in a conventional point-to-point style; As it becomes complex best practices of integrating microservices are followed that eliminate the drawbacks of point-to-point style interactions.

- Use a gateway to front all your microservices and all consumers use the microservices through the gateway only.

- No direct calls among microservices: Microservices all calls must go through the gateway.
- Micro-integration has to be done via an integration server.

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

1.3.1 14.1 Orchestration at the Microservices Layer

When multiple microservices has to be called ,to support a given business requirement another micro service built (which probably have 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.

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.

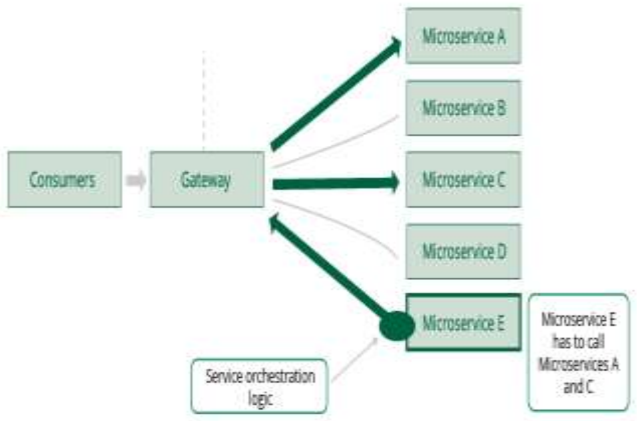


Figure 14: Service orchestration implemented at the microservices level

1.3.2 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, no need to new microservices, 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 microservices gateway implementations support this feature).

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

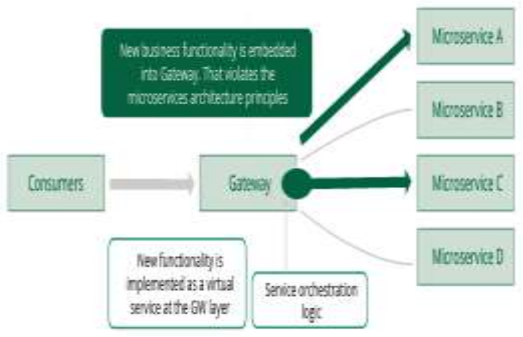

New business functionality is embedded into Gateway. That violates the microservices architecture principles
Microservice A
Microservice B
Consumers
Gateway
Microservice C
Microservice D
New functionality is implemented as a virtual service at the GW layer
Service orchestration logic

Figure Service orchestration implemented at gateway level

1.3.3 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 micro servers 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.

1.3.4 14.4 Choreography Style

In this case, there is no central component that will take care of service interactions. Various services can do pub-sub based messaging using messaging protocols.

1.4 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 microservices 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.

Key aspects in determining to incorporate an MSA in modern enterprise IT environment,

- 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

1.5 16. Conclusion

Though implementing MSA in the modern enterprise IT landscape is not a total solution. But there are quite a lot of advantages of microservices architecture or microservices. ideally, a hybrid approach of Microservices and other enterprise architectural concepts such as Integration would be more realistic.

Acknowledgments
Insert acknowledgment, if any.

References

1. http://www.ieice.org/eng/shiori/mokuji.html
2. https://www.nginx.com/resources/library/designing-deploying-microservices/
3. https://opensource.com/resources/what-docker
4. http://microservices.io/patterns/microservices.html
5. https://dzone.com/articles/microservices-design-principles
6. https://medium.com/@WSO2/guidelines-for-designing-microservices-71ee1997776c
7. https://apigee.com/about/blog/developer/12-goals-microservices
8. https://dzone.com/articles/why-container-based-deployment-is-preferred-for-mi
9. https://dzone.com/articles/microservices-in-practice-1
10. http://wso2.com/whitepapers/microservices-in-practice-key-architectural-concepts-of-an-msa/
11. https://blogs.sourceallies.com/2015/12/microservices-in-practice-challenges/
12. https://medium.com/@WSO2/microservices-in-practice-c56f4760e00

BOOKS:

1. Microservice Architecture

Building microservices with JBoss EAP 7 Babak Mozaffari
Version 1.0, June 2016

M.V.L.N. Venugopal received the M.Tech..in comp[uter science Engineerin from J..N.T.U. During 1990-1992and. M.Sc. in Electronics from O.U., respectively. During 1985-1987, he worked in TATA Info Tech Ltd. As a Sr Lead System Analyst. He has around 25 He worked in many domains including Telecom Domain. He now with PCI Private Limited