SPRING MICROSERVICES

K.RAMESH

ASPIRE Technologies

#501, 5th Floor, Mahindra Residency, Maithrivanam Road, Ameerpet, Hyderabad

Ph: 07799 10 8899, 07799 20 8899

E-Mail:ramesh@java2aspire.com **website: www.java2aspire.com**

1. INTRODUCTION

Microservice is an architecture style in which complex systems are **decomposed** into smaller services that work together to form larger business services. Microservices are autonomous, self-contained, and independently deployable.

Softwares

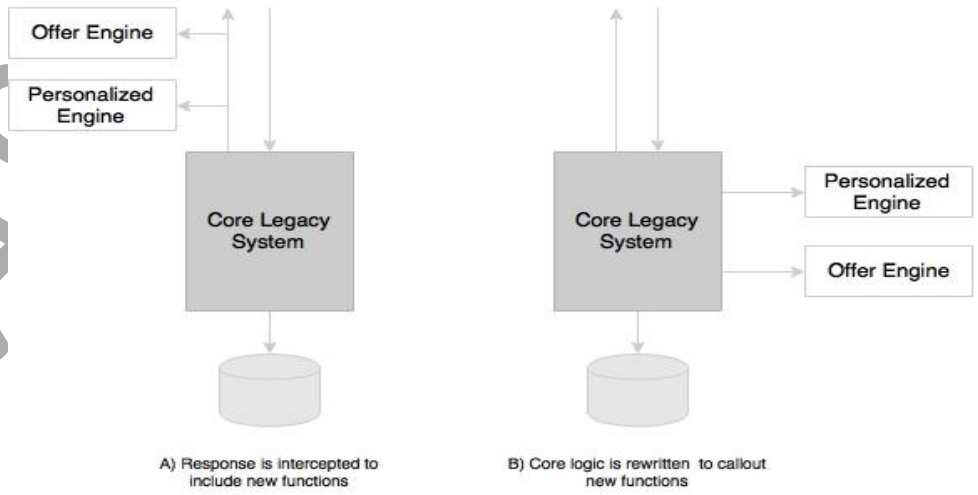
1. JDK 1.8
2. Spring Tool Suite 3.7.2 (**STS**)
3. Maven 3.x
4. Spring Framework 4.2.6.RELEASE or above
5. Spring Boot 1.3.5.RELEASE or above
6. RabbitMQ 3.5.6
7. Git
8. Spring cloud
9. Oracle DB

Microservices are not invented; rather they are evoluated from the previous architecture styles.

The evolution of microservices

The microservices evolution is greatly influenced by the **Quick business demands, Evolution of technologies** and **Evolution of Architectures** in the last few years.

Enterprisers want to quickly develop personalization engine (based on the customer's past shopping) or offers and plug them into their legacy application.



Modern architectures are expected to maximize the ability to replace their parts and minimize the cost of replacing their parts. The microservices approach is a means to achieving this.
Enterprises are no longer interested in developing consolidated applications to manage their end-to-end business functions as they did a few years ago.
Microservices promise more agility, speed of delivery, and scale compared to traditional monolithic applications resulting in less overall cost.
Emerging technologies such as Cloud, NoSQL, Hadoop, Social media, Internet of Thins (IoT), AWS, Docker, Angular JS, etc have also made us rethink the way we build software systems.
Application architecture has always been evolving alongside demanding business requirements and the evolution of technologies.
Different architecture approaches and styles such as mainframes, client server, N-tier, and service-oriented were popular at different timeframes. Irrespective of the choice of architecture styles, we always used to build **monolithic architectures**.
The microservices architecture evolved as a result of modern business demands such as agility and speed of delivery, emerging technologies, and learning from previous generations of architectures.

What are microservices?

Microservices are not invented rather many organizations such as Netflix, Amazon, and eBay successfully used the **divide-and-conquer** technique to functionally partition their **monolithic** applications into smaller **atomic** units, each performing a single function.

In below diagram, each layer holds all three business capabilities pertaining to that layer. The presentation layer has web components of all the three modules, the business layer has business components of all the three modules, and the database hosts tables of all the three modules. In most cases, **layers are physically spreadable, whereas modules within a layer are hardwired**.

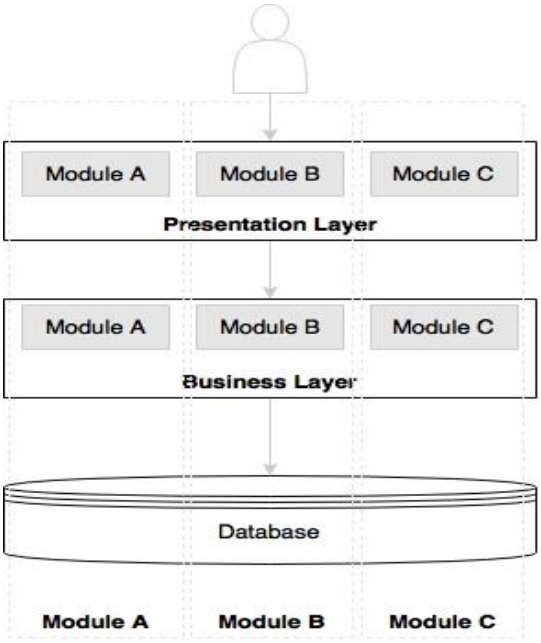


Figure: Monolithic approach

Let's now examine a microservices-based architecture. Each microservice has its own presentation layer, business layer, and database layer. Microservices are aligned towards business capabilities. By doing so, changes to one microservice doesn't impact others.

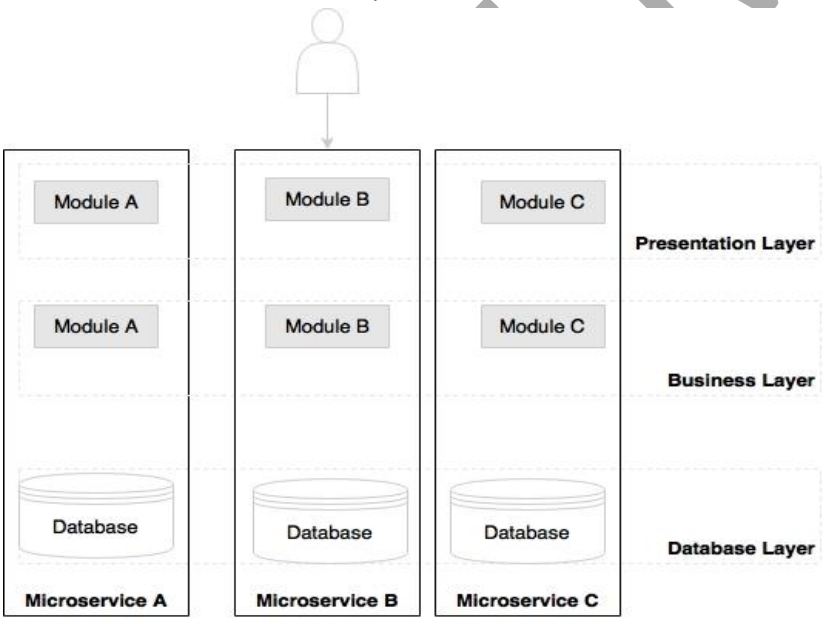


Figure: Microservices approach

What are micorservices?

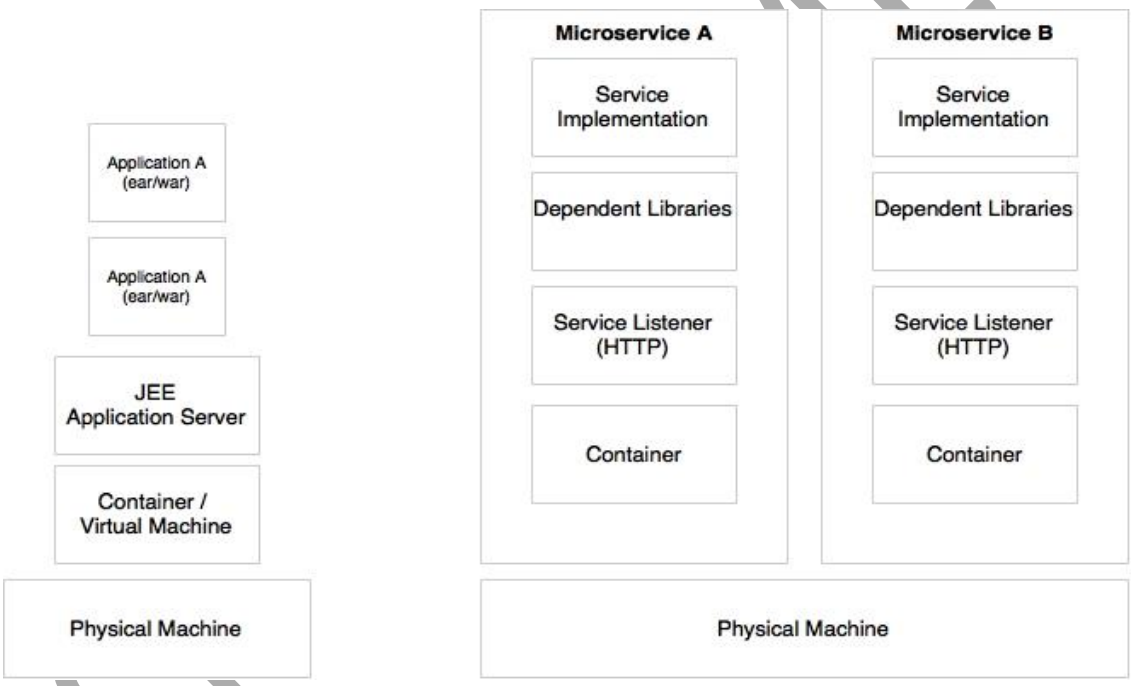
Ans) Microservices are autonomous, self-contained, loosely coupled, independently deployable and contains its own presentation layer, business layer, and database layer.

Principles of microservices

The below principles are a "**must have**" when designing and developing microservices:

1) Single microservice per single business responsibility.
2) Microservices are independently deployable. Hence they bundle all dependencies, including library dependencies, and execution environments such as web servers and containers, virtual machines and databases.

One of the major differences between Microservices and SOA is in their level of decomposition. While most SOA implementations provide service-level decomposition, microservices go further and decompose till execution environment.



In monolithic developments, we build a WAR or an EAR, then deploy it into a JEE application server, such as with JBoss, WebLogic, WebSphere, and so on. We may deploy multiple applications into the same JEE container. In microservices approach, each microservice will be built as a **fat Jar**, embedding all dependencies including web containers and run as a standalone Java process.

Characteristics of microservices

1) **Services are first class citizens**
   In the microservices architecture, there is **no more application development rather service development**. Microservices expose service endpoints as APIs and abstract all their realization details

i.e., the internal implementation logic, architecture, and technologies are completely hidden behind the service API.
Messaging (JMS / AMQP), HTTP, and REST are commonly used for interaction between microservices.
Microservices are reusable business services.
Well-designed microservices are stateless and share nothing with no shared state or conversational state maintained by the services.
Microservices are discoverable.

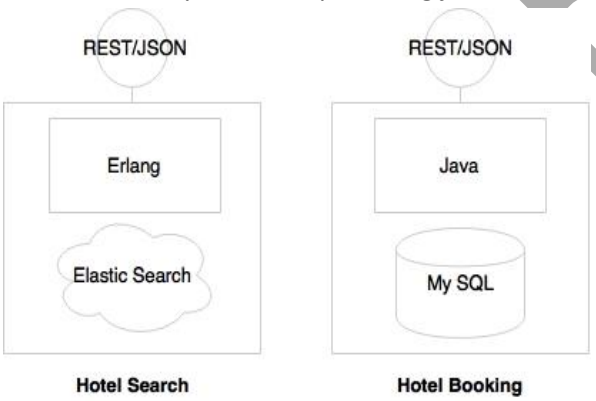
2) **Microservices are lightweight**

The microservices are aligned to a **single business capability**, so they perform only one function. When selecting supporting technologies, such as web containers, we will have to ensure that they are also lightweight. For example, Jetty or Tomcat are better choices as application containers for microservices compared to more complex traditional application servers such as WebLogic or WebSphere.

Note: Container technologies such as Docker also help us keep the infrastructure footprint as minimal as possible.

3) **Microservices with polyglot architecture**

Since Microservices are autonomous hence different services may use different technologies such as one service may be developed using java and another service may be developed using Scala, etc.



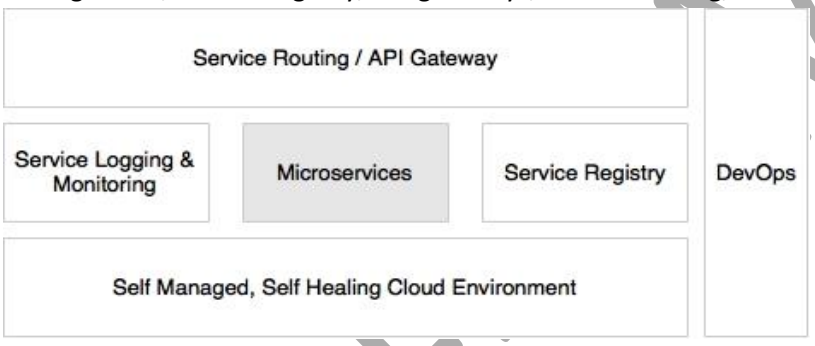
Each microservice has its own database, http container such as tomcat or jetty.

4) **Automation in a microservices environment**

As microservices break monolithic application into a number of smaller services, large enterprises may have many microservices. A large number of microservices are hard to manage until and unless automation is in place. Hence microservices should be automated from development to production: For example, automated builds, automated testing, automated deployment, and elastic scaling.

The development phase is automated using version control tools such as Git together with **Continuous Integration** (**CI**) tools such as Jenkins.
The testing phase will be automated using testing tools such as Selenium.
Automated deployments are handled using DevOps.
Infrastructure provisioning is done through Spring Cloud.

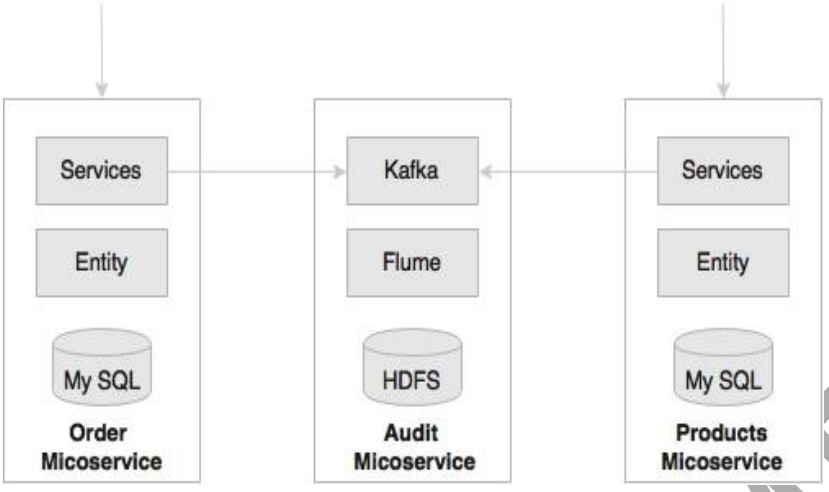
5) **Microservices with supporting ecosystem**
Microservices implementations have a supporting ecosystem including DevOps, Centralized log management, Service registry, API gateways, Service routing, Flow control mechanisms.



6) Microservices are distributed and dynamic

Microservices benefits

1) **Supports polyglot architecture**
With microservices, architects and developers can choose fit for purpose architectures and technologies for each microservice i.e., each service can run with its own architecture or technology or different versions of technologies.



2) **Enabling experimentation and innovation**

With large monolithic applications, experimentation was not easy. With microservices, it is possible to write a small microservice to achieve the targeted functionality and plug it into the system in a reactive style.

3) **Selective scaling**

A monolithic application, packaged as a single WAR or an EAR, can only be scaled as a whole. In Microservices, **each service could be independently scaled up or down depending on scalability requirement**. As scalability can be selectively applied at each service, the cost of scaling is comparatively less with the microservices approach.

4) **Allowing substitution**

Microservices are self-contained, independent deployment modules enabling the substitution of one microservice with another similar microservice. Many large enterprises follow **buy-versus-build** policies to implement software systems. A common scenario is to build most of the functions in house and buy certain niche capabilities from specialists outside.

5) **Supporting Cloud**

6) **Enabling DevOps**

Microservices are one of the key enablers of DevOps. DevOps is widely adopted as a practice in many enterprises, primarily to increase the speed of delivery and agility. DevOps advocates having agile development, high-velocity release cycles, automatic testing, and automated deployment.

Relationship with SOA

SOA and Microservices follow similar concepts i.e., many service characteristics are common in both approaches.

Service-Oriented Architecture (SOA) is an architectural style that supports service orientation. Service orientation is a way of thinking in terms of services and service-based development and the outcomes of services.

A service: Is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data, and consolidate drilling reports)

1) It is self-contained.
2) It may be composed of other services.
3) It is a "black box" to consumers of the service."

We observed similar aspects in microservices as well. Then how microservices differ from SOA?

One of the major differences between Microservices and SOA is in their level of abstraction. While most SOA implementations provide service-level abstraction, Microservices go further and abstract the realization and execution environment i.e., in SOA development, we may deploy multiple services into the same JEE container. In the microservices approach, each microservice will be built as a **fat Jar**, embedding all dependencies including web containers and run as a standalone Java process.

In case of Legacy modernization, the services are built and deployed in the ESB layer connecting to backend systems using ESB adapters. In these cases, microservices are different from SOA.

Microservice use cases

A microservice will not solve all the architectural challenges of today's world. There is no hard-and-fast rule or rigid guideline on when to use microservices.

The first and the foremost activity is to do a test of the use case against the microservices benefits.

Let's discuss some commonly used scenarios that are suitable candidates for a microservices architecture:

1) Migrating a monolithic application due to improvements required in scalability, manageability, agility, or speed of delivery.
2) Utility computing scenarios such as integrating an optimization service, forecasting service, price calculation service, prediction service, offer service, recommendation service, and so on are good candidates for microservices because these are independent stateless computing units that accept certain data, apply algorithms, and return the results.
3) Independent technical services such as the communication service, the encryption service, authentication services, and so on are also good candidates for microservices.
4) In many cases, we can build headless business applications or services that are autonomous in nature—for instance, the payment service, login service, flight search service, customer profile service,

notification service, and so on. These are normally reused across multiple channels and, hence, are good candidates for building them as microservices.

5) There could be micro or macro applications that serve a single purpose and performing a single responsibility.
6) Highly agile applications, applications demanding speed of delivery or **time to market**, innovation pilots, applications selected for DevOps, applications of the System of Innovation type, and so on could also be considered as potential candidates for the microservices architecture.
7) Applications that we could anticipate getting benefits from microservices such as polyglot requirements.

There are few scenarios in which we should consider avoiding microservices:

1) If the organization's policies are forced to use centrally managed heavyweight components such as ESB to host a business logic.
2) If the organization's culture, processes, and so on are based on the traditional waterfall delivery model, lengthy release cycles, manual deployments and cumbersome release processes, no infrastructure provisioning, and so on, then microservices may not be the right fit.

Microservices early adopters

Many organizations have already successfully embarked on their journey to the microservices world.

1) Netflix

Netflix, an international on-demand media streaming company, is a pioneer in the microservices space. Netflix transformed their large pool of developers developing traditional monolithic code to smaller development teams producing microservices. At Netflix, engineers started with monolithic, went through the pain, and then broke the application into smaller units that are loosely coupled and aligned to the business capability.

2) Amazon

The well-architected monolithic application was based on a tiered architecture with many modular components. However, all these components were tightly coupled. As a result, Amazon was not able to speed up their development cycle. Amazon then separated out the code as independent functional services, wrapped with web services, and eventually advanced to microservices.

3) Twitter

When Twitter experienced growth in its user base, they went through an architecture-refactoring cycle. With this refactoring, Twitter moved away from a typical web application to an API-based event driven code. Twitter uses Scala and Java to develop microservices with polyglot persistence.

4) Uber

When Uber expanded their business from one city to multiple cities, the challenges started. Uber then moved to microservice based architecture by breaking the system into smaller independent units. Each module was given to different teams and empowered them to choose their language, framework, and database. Uber has many microservices deployed in their ecosystem using REST.

…

Building microservices with boot

Traditionally a war was explicitly created and deployed on a Tomcat server. But microservices need to develop services as executables, self-contained JAR files with an embedded HTTP listener (such as tomcat or jetty). Spring boot is a tool to develop such kinds of services. Spring Boot also enables microservices development by packaging all the required runtime dependencies in a **fat executable JAR file**.

2. DESIGNING MICROSERVICES

Microservices have gained enormous popularity in recent years. They have evolved as the preferred choice of architects, putting SOA into the backyards. While acknowledging the fact that microservices are a vehicle for developing scalable cloud native systems, **successful microservices need to be carefully designed** to avoid catastrophes. Hence number of factors are to be considered when designing microservices, as detailed in the following sections.

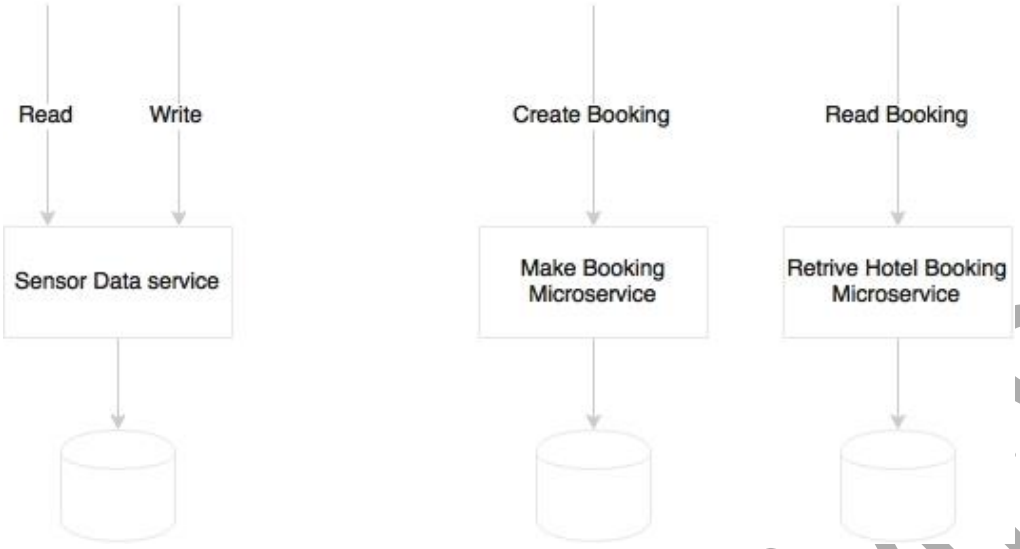
Identifying microservice boundaries

The following scenarios could help in defining microservice boundaries:

1) Autonomous functions : If the function under review is autonomous by nature, then it can be taken as a microservices boundary.
2) Size of deployable unit: A good microservice ensures that the size of its deployable units remains manageable.
3) Polyglot Architecture: If different modules need different architectures, different technologies, etc then split them as separate Microservices.
4) Selective Scaling: All functions may not require the same level of scalability. Sometimes it may be appropriate to determine boundaries based on scalability requirements. For example, in the hotel booking, the Search microservice has to scale considerably more than Booking microservice.
5) Small, Agile teams
6) Single Responsibility: One microservice per one business capability.
7) Replicability or Changeability: Microservices boundaries should be identified in such a way that each microservice is easily detachable from the overall system.
8) Coupling and Cohesion

Number Of Endpoints for a Microservice

The number of endpoints is not really a decision point. In some cases, there may be only one endpoint, whereas in some other cases, there could be more than one endpoint in a microservice.



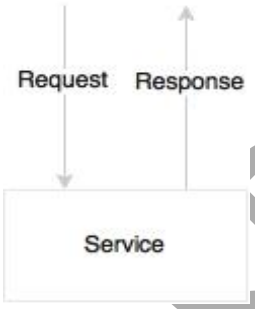
The Sensor data service has two logical end points: read and write

Communication styles

Communication between microservices can be designed either in **synchronous** or **asynchronous** styles.

Synchronous style

The following diagram shows an example of synchronous (request/response) style service:



In synchronous communication, there is no shared state or queue. When a caller requests a service, it passes the required information and **waits** for a response.

Advantages:

1) No messaging server overhead.
2) The error will be propagated back to the caller immediately.

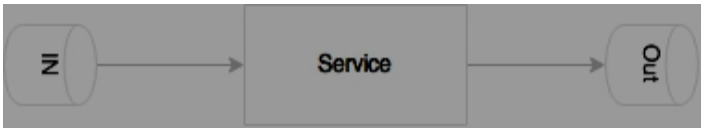
Dis advantages:

1) The caller has to wait until the requested process gets completed.

2) Adds hard dependencies between Microservices i.e., If one service in the chain fails, then the entire service chain will fail.

Asynchronous style

The following diagram is a service designed to accept an asynchronous message as input, and send the response asynchronously for others to consume:



The asynchronous style is based on **reactive** event loop semantics which **decouple** microservices.

Advantages:

1) Decouple Microservices
2) Higher level of scalability because of services are independent. Hence if there is a slowdown in one of the services, it will not impact the entire chain.

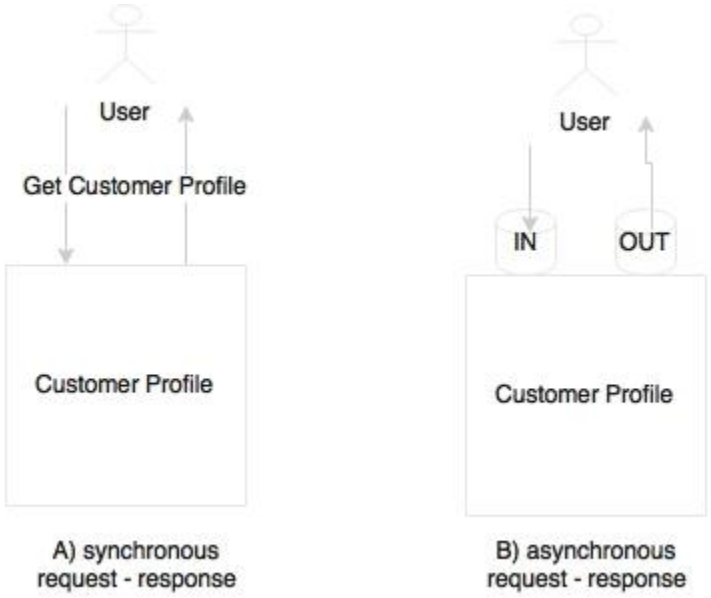
Dis advantages:

1) It has a dependency to an external messaging server.
2) It is complex to handle the fault tolerance of a messaging server.

How to decide which style to choose?

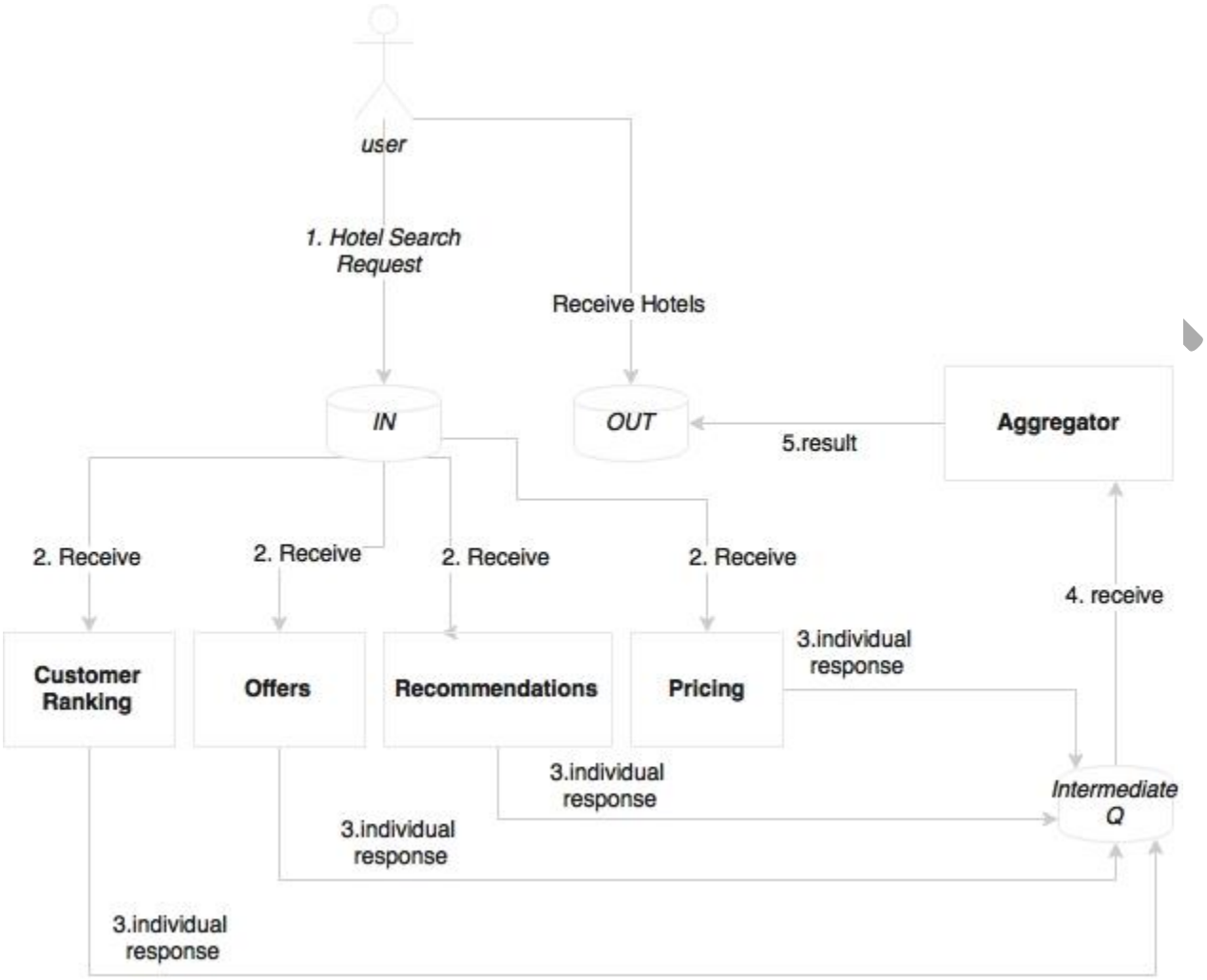
It is not possible to develop a system with just one approach. A combination of both approaches are required based on the use cases. In principle, the asynchronous approach is great for building true, scalable microservice systems. However, attempting to model everything as asynchronous leads to complex system designs.

How does the following example look in the context where an end user clicks on a UI to get profile details?



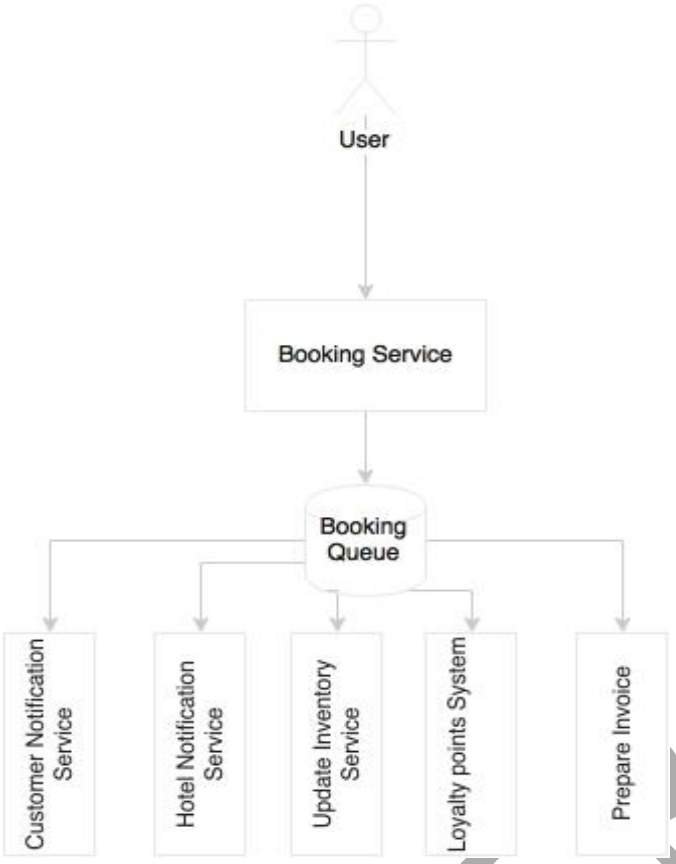
This is perfect scenario for synchronous communication. This can also be modeled in an asynchronous style by pushing a message to an input queue, and waiting for a response in an output queue till a response is received for the given correlation ID. However, though we use asynchronous messaging, the user is still blocked for the entire duration of the query. Hence no advantage of using asynchronous style.

Another use case is user clicking on a UI to search hotels, which is depicted in the following diagram:



When the system receives this request, it calculates the customer ranking, gets offers based on the destination, gets recommendations based on customer preferences, and optimizes the prices based on customer values and revenue factors, and so on. In this case, we have an opportunity to do many of these activities in parallel so that we can aggregate all these results before presenting them to the customer. As shown in the preceding diagram, virtually any computational logic could be plugged in to the search pipeline listening to the **IN** queue. An effective approach in this case is to start with a synchronous request response, and refactor later to introduce an asynchronous style when there is value in doing that.

The following example shows a fully asynchronous style of service interactions:



When booking is successful, it sends a message to the customer's e-mail address, sends a message to the hotel's booking system, updates the cached inventory, updates the loyalty points system, prepares an invoice, and perhaps more. Instead of pushing the user into a long wait state, a better approach is to break the service into pieces. Let the user wait till a booking record is created by the Booking Service. On successful completion, a booking event will be published, and return a confirmation message back to the user. Subsequently, all other activities will happen in parallel, asynchronously.

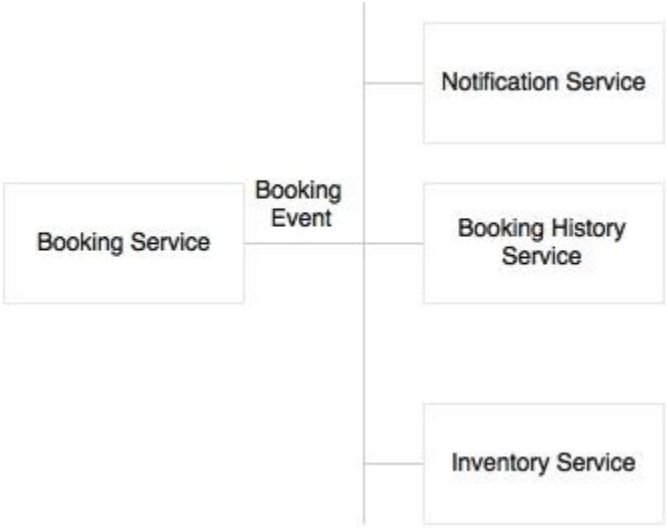
Conclusion:

In general, an asynchronous style is always better in the microservices world, but identifying the right pattern should be purely based on merits. If there are no merits in modeling a communication in an asynchronous style, then use the synchronous style till we find an appealing case.

Orchestration of Microservices

Composability (means controlling) is one of the service design principles. In the SOA world, ESBs are responsible for composing a set of fine-grained services i.e., In the SOA world, ESBs play the role of orchestration.

Microservices are autonomous. This means that all required components to complete their function should be within the service. This includes the database, orchestration of its internal services, state management, and so on. But in reality, microservices may need to talk with other microservices to fulfil their function.
The following approach is preferred to connect multiple microservices together:



Number Of VMs per MicroService

The one microservice can be deployed in one or multiple Virtual Machines (VMs) by replicating the deployment for scalability and availability.
Multiple Microservices can be deployed in one VM if the service is simple and the traffic volume is less.

In case of cloud infrastructure, the developers need not to worry about where the services are running. Developers may not even think about capacity planning. Services will be deployed in a compute cloud. Based on the infrastructure availability and the nature of the service, the infrastructure self-manages deployments.

Can microservices share data stores?

In principle, microservices should abstract presentation, business logic, and data stores i.e., each microservice logically could use an independent database.
Shared data models, Shared schema, and shared tables are disasters when developing microservices.
If the services have only a few tables, it may not be worth investing a full instance of a database like Oracle instance. In such cases, schema level segregation is good enough to start with. The ideal scenario is to use local transactions within a microservice if required, and completely avoid distributed transactions.

Shared Libraries

Sometimes code and libraries may be duplicated in order to adhere to autonomous and self-contained principle.

The eligibility for a flight upgrade will be checked at the time of check-in as well as when boarding. This was the trade-off between overheads in communication versus duplicating libraries in multiple services:

1) It may be easy to duplicate code or shared library but downside of this approach is that in case of a bug or an enhancement on the shared library, it has to be upgraded in more than one place.

2) An alternative option of developing the shared library as another microservice itself needs careful analysis. If it is not qualified as a microservice from the business capability point of view, then it may add more complexity than its usefulness.

3. MICROSERVICES CHALLENGES

In this chapter, we will review some of the challenges with microservices, and how to address them for a successful microservice development.

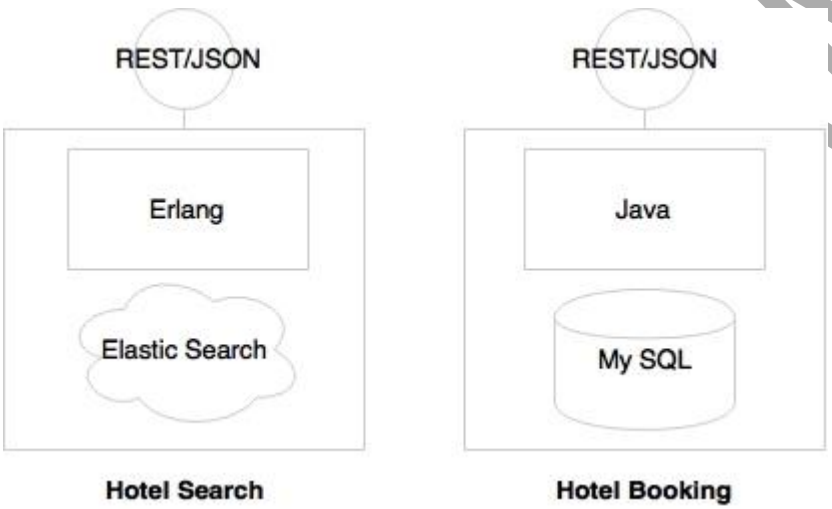
Infrastructure provisioning

With many Microservices running, manual deployment could lead to significant operational overheads and the chances of errors are high.

To address this challenge, Microservices should use elastic cloud-like infrastructure which can automatically provision VMs or containers, automatically deploy applications, adjust traffic flows, replicate new version to all instances, and gracefully phase out older versions. The automation also takes care of scaling up elastically by adding containers or VMs on demand, and scaling down when the load falls below threshold.

Data Islands

Microservices use their own local transactional store, which is used for their own transactional purposes.



In the preceding diagram, Hotel search is expected to have high transaction volume hence preferred to use Elasticsearch. The Hotel booking needs more ACID transactions hence preferred to use MySQL. That means different Microservices may use different types of databases which leads data islands.

What if we want to do an analysis by combining data from two data stores?

In order to satisfy this requirement, a data warehouse (traditional) or a **data lake** is required. The tools like Spring Cloud Data Flow, Kafka, Flume, etc are useful.

Logging and monitoring

Since each microservice is deployed independently, they emit separate log files. This makes it extremely difficult to debug and understand the behavior of the services through logs. Hence we need **centralized logging** mechanism which can be achieved using Graylog, Splunk, ELK stack, AWS CloudTrail, Google Cloud Logging, Spring Boot's Loggly , Logstash, Kibana, Elastic search, SaaS, etc.

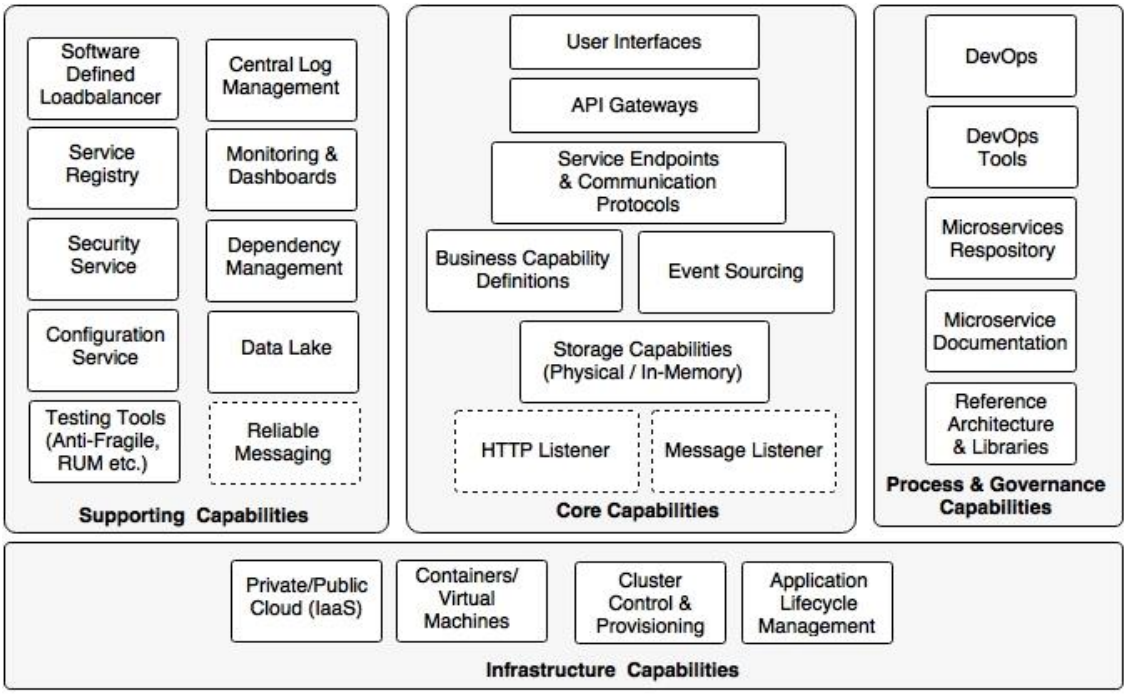
Organization culture

One of the biggest challenges in microservices implementation is the organization culture. To harness the speed of delivery of microservices, the organization should adopt Agile development processes, continuous integration, automated QA checks, automated delivery pipelines, automated deployments, and automatic infrastructure provisioning.

Organizations following a waterfall development or heavyweight release management processes with infrequent release cycles are a challenge for microservices development. Insufficient automation is also a challenge for microservices deployment.

4. THE MICROSERVICES CAPABILITY MODEL

We will review a capability model for microservices based on the design guidelines, challenges, common patterns and solutions described so far.



The capability model is broadly classified into four areas:

1. **Core capabilities**: These are part of the microservices themselves
2. **Supporting capabilities**: These are software solutions supporting core microservice implementations
3. **Infrastructure capabilities**: These are infrastructure level expectations for a successful microservices implementation
4. **Governance capabilities**: These are more of process, people, and reference information

Core capabilities

The core capabilities are explained as follows:

- **Service listeners** (HTTP/Message): If microservices are enabled for a HTTP-based service endpoint, then the **HTTP listener is embedded within the microservices**, thereby eliminating the need to have any external application server requirement.
  If the microservice is based on asynchronous communication, then instead of an HTTP listener, a **message listener** is started. Spring Boot and Spring Cloud Streams provide this capability.

- **Storage capability**: The microservices have some kind of storage mechanisms to store state or transactional data pertaining to the business capability. The storage could be either a physical storage

(RDBMS such as MySQL; NoSQL such as Hadoop, Cassandra, Neo 4J, Elasticsearch, and so on), or it could be an in-memory store (cache like Ehcache, Redis, data grids like Hazelcast, Infinispan, and so on)

- **Business capability definition:** This is the core of microservices, where the business logic is implemented. This could be implemented in any applicable language such as Java, Scala, Conjure, Erlang, and so on. All required business logic to fulfill the function will be embedded within the microservices themselves.

- **Event sourcing:** Microservices send out state changes to the external world without really worrying about the targeted consumers of these events. These events could be consumed by other microservices, audit services, replication services, or external applications, and the like. This allows other microservices and applications to respond to state changes.

- **Service endpoints and communication protocols:** These define the APIs for external consumers to consume. These could be synchronous endpoints or asynchronous endpoints.

- **API gateway:** The API gateway provides a level of indirection by either proxying service endpoints or composing multiple service endpoints. There are many API gateways available in the market. Spring Cloud Zuul, Mashery, Apigee, and 3scale are some examples of the API gateway providers.

- **User interfaces:** Generally, user interfaces are also part of microservices for users to interact with the business capabilities realized by the microservices. These could be implemented in any technology.

Infrastructure capabilities

Certain infrastructure capabilities are required for a successful deployment, and managing large scale microservices. When deploying microservices at scale, not having proper infrastructure capabilities can be challenging, and can lead to failures:

- **Cloud:** Microservices implementation is difficult in a traditional data center environment with long lead times to provision infrastructures. Even a large number of infrastructures dedicated per microservice may not be very cost effective. Managing them internally in a data center may increase the cost of ownership and cost of operations. A cloud-like infrastructure is better for microservices deployment.

- **Containers or virtual machines:** Managing large physical machines is not cost effective, and they are also hard to manage. Virtualization is adopted by many organizations because of its ability to provide optimal use of physical resources. It also provides resource isolation. It also reduces the overheads in managing large physical infrastructure components. Containers are the next generation of virtual machines. VMWare, Citrix, and so on provide virtual machine technologies. Docker, Drawbridge, Rocket, and LXD are some of the containerizer technologies.

- **Cluster control and provisioning**: Once we have a large number of containers or virtual machines, it is hard to manage and maintain them automatically. Cluster control tools provide a uniform operating environment on top of the containers, and share the available capacity across multiple services. Apache Mesos and Kubernetes are examples of cluster control systems.

- **Application lifecycle management**: Application lifecycle management tools help to invoke applications when a new container is launched, or kill the application when the container shuts down. Application life cycle management allows for script application deployments and releases. It automatically detects

failure scenario, and responds to those failures thereby ensuring the availability of the application. This works in conjunction with the cluster control software. Marathon partially addresses this capability.

Supporting capabilities

Supporting capabilities are not directly linked to microservices, but they are essential for large scale microservices development:

- **Software defined load balancer**: The load balancer should be smart enough to understand the changes in the deployment topology, and respond accordingly. This moves away from the traditional approach of configuring static IP addresses, domain aliases, or cluster addresses in the load balancer. When new servers are added to the environment, it should automatically detect this, and include them in the logical cluster by avoiding any manual interactions. Similarly, if a service instance is unavailable, it should take it out from the load balancer. A combination of *Ribbon*, Eureka, and Zuul provide this capability in Spring Cloud Netflix.

- **Central log management**: A capability is required to centralize all logs emitted by service instances with the correlation IDs. This helps in debugging, identifying performance bottlenecks, and predictive analysis. The result of this is fed back into the life cycle manager to take corrective actions.

- **Service registry**: A service registry provides a runtime environment for services to automatically publish their availability at runtime. A registry will be a good source of information to understand the services topology at any point. *Eureka* from Spring Cloud, Zookeeper, and Etcd are some of the service registry tools available.

- **Security service**: A distributed microservices ecosystem requires a central server for managing service security. This includes service authentication and token services. OAuth2-based services are widely used for microservices security. Spring Security and *Spring Security OAuth* are good candidates for building this capability.

- **Service configuration**: All service configurations should be externalized as discussed in the Twelve-Factor application principles. A central service for all configurations is a good choice. *Spring Cloud Config server*, and Archaius are out-of-the-box configuration servers.

- **Testing tools (anti-fragile, RUM, and so on)**: Netflix uses Simian Army for anti-fragile testing. Matured services need consistent challenges to see the reliability of the services, and how good fallback mechanisms are. Simian Army components create various error scenarios to explore the behavior of the system under failure scenarios.

- **Monitoring and dashboards**: Microservices also require a strong monitoring mechanism. This is not just at the infrastructure-level monitoring but also at the service level. Spring Cloud Netflix Turbine, Hysterix Dashboard, and the like provide service level information. End-to-end monitoring tools like AppDynamic, New Relic, Dynatrace, and other tools like statd, Sensu, and Spigo could add value to microservices monitoring.

- **Dependency and CI management**: We also need tools to discover runtime topologies, service dependencies, and to manage configurable items. A graph-based CMDB is the most obvious tool to manage these scenarios.

- **Data lake**: We need a mechanism to combine data stored in different microservices, and perform near real-time analytics. A data lake is a good choice for achieving this. Data ingestion tools like Spring Cloud Data Flow, Flume, and Kafka are used to consume data. HDFS, Cassandra, and the like are used for storing data.

- **Reliable messaging**: If the communication is asynchronous, we may need a reliable messaging infrastructure service such as RabbitMQ or any other reliable messaging service. Cloud messaging or messaging as a service is a popular choice in Internet scale message-based service endpoints.

Process and governance capabilities

The last piece in the puzzle is the process and governance capabilities that are required for microservices:

- **DevOps**: The key to successful implementation of microservices is to adopt DevOps. DevOps compliment microservices development by supporting Agile development, high velocity delivery, automation, and better change management.

- **DevOps tools**: DevOps tools for Agile development, continuous integration, continuous delivery, and continuous deployment are essential for successful delivery of microservices. A lot of emphasis is required on automated functioning, real user testing, synthetic testing, integration, release, and performance testing.

- **Microservices repository**: A microservices repository is where the versioned binaries of microservices are placed. These could be a simple Nexus repository or a container repository such as a Docker registry.

- **Microservice documentation**: It is important to have all microservices properly documented. Swagger or API Blueprint are helpful in achieving good microservices documentation.

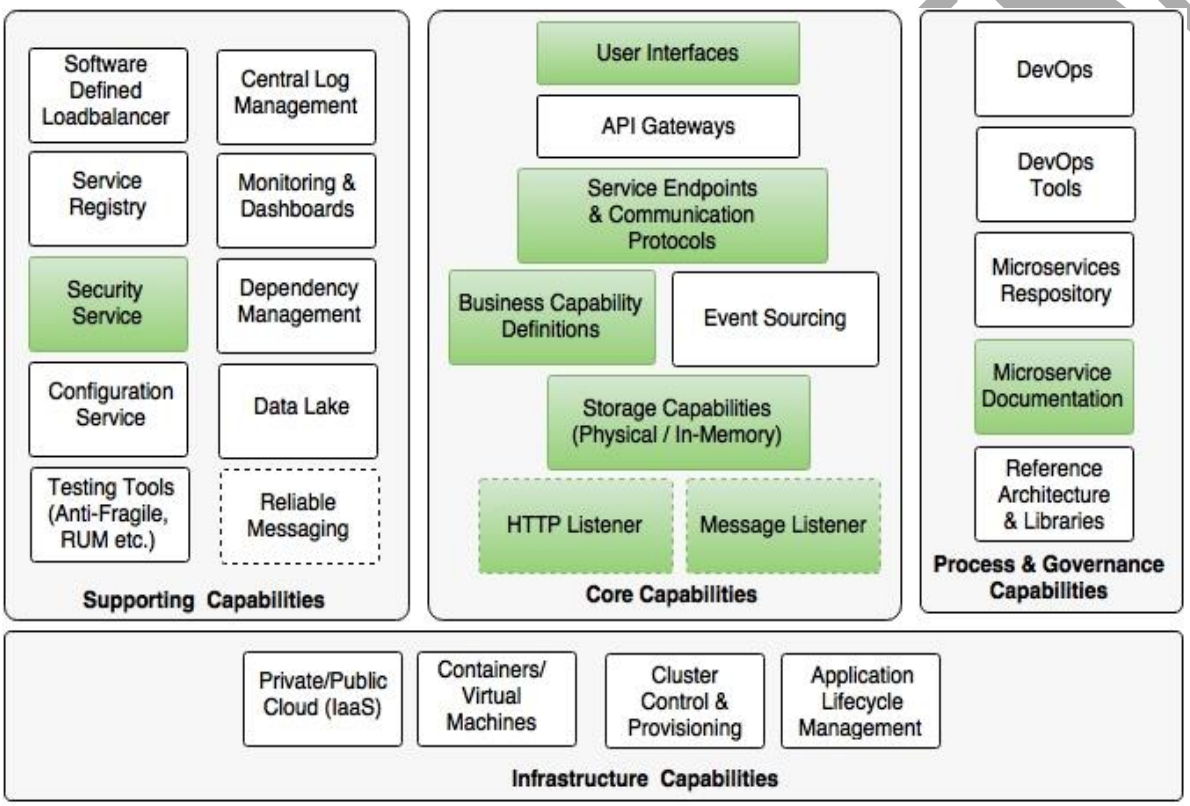
- **Reference architecture and libraries**: The reference architecture provides a blueprint at the organization level to ensure that the services are developed according to certain standards and guidelines in a consistent manner. Many of these could then be translated to a number of reusable libraries that enforce service development philosophies.

5. MICROSERVICES EVOLUTION – A CASE STUDY

We will discuss BrownField Airline and their journey from a monolithic **Passenger Sales and Service** (PSS) application to a next generation microservices architecture by adhering to the principles and practices that were discussed before.

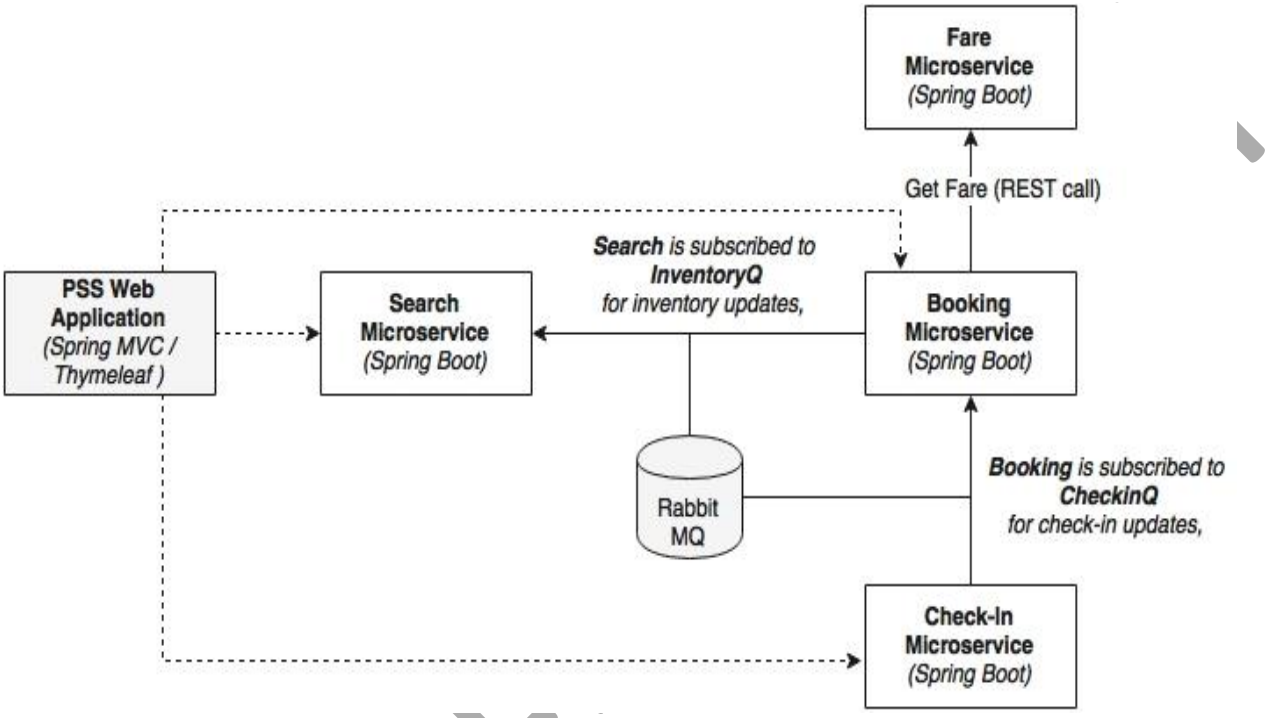
Reviewing the microservices capability model

In this chapter, we will explore the following microservices capabilities highlighted in green color from the microservices capability model discussed before.



We are implementing four microservices such as Fare, Search, Booking, and Check-in. In order to test the application, there is a website application developed using Spring MVC with Thymeleaf templates (needed for html pages). The asynchronous messaging is implemented with the help of RabbitMQ. In this implementation, the **Oracle database is used with separate schema for each Microservice**.

The code in this section demonstrates all the capabilities highlighted in green color above.



Each service has multiple packages and their purposes are explained as follows:

1. The entity package contains the JPA entity classes for mapping to the database tables.
2. The repository package contains repository classes, which are based on Spring Data JPA.
3. The component package hosts all the service components where the business logic is implemented.
4. The controller package hosts the **REST endpoints** and the **Messaging endpoints**. Controller classes internally utilize the component classes for execution.
5. The root package (com.brownfield.pss.fares) contains the default Spring Boot application.

The below table contains service endpoints and communication styles:

Microservice Name	REST endpoints (synchronous)	Messaging Endpoints (asynchronous)	Used By
FareFlightTickets	http://localhost:8081/fares/get		Booking microservice
SearchFlightTickets	http://localhost:8090/search/get		Website
SearchFlightTickets		@RabbitListener(queues = "**SearchQ**")	Search microservice itself subscribed to **SearchQ** for inventory updates.
BookingFlightTickets	http://localhost:8060/booking/create		Website

BookingFlightTickets	http://localhost:8060/booking/get/{id}		Checkin, website
BookingFlightTickets		template.convertAndSend("**SearchQ**", message);	Search Microservice
BookingFlightTickets		@RabbitListener(queues = "**CheckINQ**")	Booking service subscribed to **CheckINQ** for check-in updates.
CheckInCustomers	http://localhost:8070/checkin/create		Website
CheckInCustomers	http://localhost:8070/checkin/get/{id}		Not used
CheckInCustomers		template.convertAndSend("**CheckINQ**", message);	Booking Microservice

The following steps are used to setup PSS Microservices project:

1) Create tablespaces, schemas, tables, sequences and insert data by referring 'Documents/Misc/ Airline_PSS_Schema.doc' file.
2) Start STS (Spring Tool Suite) and select '**MicroservicesWorkspace**' from the backup.
   Note: Download STS from https://spring.io/tools/sts/all
3) Start FareFlightTickets by right click and Run as **Spring Boot App**.
4) Install **RabbitMQ** Server from Softwares folder. After installation check service status in start-> run -> services.msc
   Observation: Status: Running, Startup type: Automatic
   Note: The pre-requisite for RabbitMQ is **Erlang**. (Install OTP_win64_19.3.exe from softwares folder)
5) Start SearchFlightTickets by right click and Run as **Spring Boot App**.
6) Start BookingFlightTickets by right click and Run as Spring Boot App.
7) Start CheckInCustomers by right click and Run as Spring Boot App.
8) Start FlightWebSite by right click and Run as Spring Boot App.

We have accomplished the following items in our microservice implementation so far:

1. Each microservice exposes a set of REST/JSON endpoints for accessing business capabilities
2. Each microservice implements certain business functions using the Spring framework.
3. Each microservice has its own schema in Oracle database.
4. Microservices are built with Spring Boot, which has an embedded Tomcat server as the HTTP listener.
5. RabbitMQ is used as an external messaging service. Search, Booking, and Check-in interact with each other through asynchronous messaging.
6. An OAuth2-based security mechanism is developed to protect the Microservices.