**SPRING MICROSERVICES**

**Microservices** is an architectural approach in software development, where a large application is broken down into smaller, **independent**, and **loosely coupled** modules or services. Each of these services corresponds to a unique business functionality and can operate independently.

## Key aspects of microservices include:

**Independent Deployment:** Each microservice can be deployed, upgraded, scaled, and restarted independently of other services in the application.

**Single Responsibility**: Following the single responsibility principle, each microservice focuses on a single functionality.

**Decentralized Data Management:** Each microservice has its own dedicated database or data store, which results in decentralized data management.

**Distributed Development:** Microservices can be built using different programming languages and can be managed by different teams.

**Inter-Service Communication:** Microservices communicate with each other through well-defined APIs and protocols, such as REST or messaging queues.

**Isolation of Failures:** If one service fails, the others can continue to function. This isolation improves the overall resilience of the application.

**Automation:** Microservices are typically developed with automation in mind, including automated testing, continuous integration, and continuous deployment.

Overall, the microservices architecture provides flexibility, scalability, and quicker time to market, but it also introduces complexity in terms of managing, deploying, and monitoring distributed systems. It's a good fit for large, complex, and evolving applications, particularly those that need to handle high levels of traffic and data.

Top of Form

Bottom of Form

## Software

**JDK 17**: JDK stands for Java Development Kit. JDK 17 is a long-term support (LTS) version of Java, meaning that it will receive updates and support for an extended period.

**Spring Tool Suite 4x (STS) / Eclipse with STS plug-in, IntelliJ Idea**: These are integrated development environments (IDEs) for Java. STS is specifically tailored for Spring applications, and IntelliJ IDEA is a versatile IDE that supports many languages and frameworks, including Spring.

**Maven 3.x/ Gradle**: These are build automation tools used in Java applications. They help manage dependencies, compile source code, run tests, and package the application for distribution.

**Spring Framework 6x RELEASE or above**: The Spring Framework is a popular Java framework for developing enterprise-grade applications. It provides a comprehensive programming and configuration model and a wide range of features.

**Spring Boot 3.1.x**: Spring Boot is a project that builds on the Spring Framework, aiming to simplify the setup and development of Spring applications. It provides production-ready defaults to minimize the configuration typically involved with Spring applications.

**RabbitMQ / Apache Kafka**: These are messaging systems. RabbitMQ is a message broker that supports multiple messaging protocols. Apache Kafka is a distributed event streaming platform used for high-performance real-time data pipelines, streaming analytics, data integration, and mission-critical applications.

**MySQL/MongoDB/Redis**: These are databases. MySQL is a relational database, MongoDB is a NoSQL database, and Redis is an in-memory data structure store used as a database, cache, and message broker.

**Postman for testing endpoints**: Postman is a platform for API development, and it's often used for testing API endpoints. It provides a GUI for constructing requests and reading responses.

**Kubernetes**: Kubernetes is an open-source platform designed to automate deploying, scaling, and operating application containers. It groups containers that make up an application into logical units for easy management and discovery.

## Monolithic application and Its disadvantages

A monolithic application is a software architecture where the application is built as a single, autonomous unit. While monolithic applications have some benefits, such as **simplicity in development and testing**, there are several issues that can arise with this architecture:

* **Scalability**: Monolithic applications can be harder to scale compared to microservices. With a monolithic architecture, you generally have to scale the entire application, not just the parts experiencing high load. This can be less efficient and more expensive.
* **Development speed and complexity**: As the application grows, the codebase becomes larger and more complex. This can slow down development, as it becomes harder to understand and make changes to the code. It can also increase the risk of bugs.
* **Technological limitations**: With a monolithic application, you're usually locked into the technology stack you started with. It's challenging to adopt new technologies or frameworks because it requires changes throughout the entire codebase.
* **Deployment risks**: Since the application is a single unit, even a small change in the codebase requires a full redeployment. If there's a problem during deployment, it could impact the entire application, leading to more significant downtime.
* **Fault isolation**: In a monolithic architecture, if there's a bug or an issue with one part of the application, it could potentially bring down the entire system. This is different from a microservices architecture, where an issue in one service doesn't necessarily impact others.
* **Dependency issues**: A large monolithic application tends to have intertwined dependencies, making it difficult to modify or update components without affecting others.

Bottom of Form

## What is Java microservices?

**Java microservices** are an approach to developing applications where each component is built as an independent service, written using the Java programming language. These microservices typically embody the following principles:

**Single Responsibility**: Each microservice is designed to do one thing well, according to the single responsibility principle. This improves the maintainability and comprehensibility of each service.

**Independence**: Each Java microservice can be developed, deployed, and scaled independently of the others. This allows different parts of an application to evolve at their own pace.

**Decentralized Data Management**: Each microservice has its own separate database or data store, promoting data consistency and integrity.

**Distributed Development**: Given their independence, different teams can work on different microservices, potentially using different technologies and practices best suited for their specific microservice.

**Inter-Service Communication**: The microservices interact with each other through well-defined APIs, often leveraging protocols like HTTP/REST or messaging queues.

Java, being a robust, versatile, and platform-independent language, provides several powerful frameworks for creating microservices. These include **Spring Boot**, **Micronaut**, **Quarkus**, **Dropwizard**, and others.

These Java microservices can be individually packaged into containers, and then orchestrated using technologies like Docker and Kubernetes, providing a powerful and scalable system. This architectural style is especially beneficial for large, complex applications requiring high scalability and flexibility, where different services can be modified, updated, and deployed independently.

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. reframe words with good sentenses

****

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 don’t impact others.



Figure: Microservices approach

## What scenario to use Monolithic and Microservices?

Monolithic applications and microservices both have their own pros and cons, and are better suited to different scenarios. Here are some situations where you might want to choose one over the other:

## Best Scenarios for Monolithic Applications:

**Small Applications**: If the scope of your application is small to medium, a monolithic architecture could be more suitable. With less complexity, the benefits of microservices might not outweigh the overhead of setting them up and managing them.

**Simple Domains**: If your business logic is not very complex and you don't expect it to change drastically or frequently, a monolithic application can be a good fit.

**Rapid Development and MVPs**: If you need to rapidly develop and deploy an application, maybe for a minimum viable product (MVP), a monolithic architecture allows for faster development and simpler deployment.

**Single Team Development**: If you have a small team where everyone can understand the whole system, the simplicity of a monolithic architecture can be an advantage.

## Best Scenarios for Microservices:

1. **Large Applications and Organizations**: If your application is large and complex, or if your organization is large with many development teams, microservices can be a good choice. Each team can focus on a different service, using the technology stack that is best suited to that service's requirements.
2. **Scalability Needs**: If different parts of your application need to scale differently, microservices allow you to scale only the services that need it, which can be more efficient and cost-effective.
3. **Frequent Updates and Changes**: If you have parts of your application that need to be updated or changed frequently, microservices can be beneficial. You can update a single service without having to redeploy the entire application.
4. **Resilience**: If it's crucial that a failure in one part of the application doesn't bring down the whole system, microservices provide isolation between services.
5. **Technological Flexibility**: If you want to use different technologies, languages, or frameworks in different parts of your application, a microservices architecture can allow that.

It's important to note that the choice between monolithic and microservices architectures isn't binary. There are architectural patterns that fall in-between, like modular monoliths, that can provide some of the benefits of both. The decision should be based on your specific requirements and constraints.

Top of Form

Bottom of Form

## Principles of microservices

The principles of microservices are guidelines that dictate how microservices-based applications are designed and function. They help to ensure that applications are scalable, maintainable, and reliable. Here are some key principles:

1. **Single Responsibility**: Each microservice should focus on a single function or process that relates to a specific business capability.
2. **Independence**: Each microservice should be able to operate independently. This means it can be updated, deployed, and scaled without affecting the operation of other services.
3. **Decentralization**: All aspects, including data management and configuration, should be decentralized. This is often accomplished by giving each microservice its own database or data store.
4. **Autonomy**: Microservices should be autonomous, meaning they should be self-contained and independent of other services. They should be able to function correctly even if other services are unavailable.
5. **Failure Isolation**: The failure of one service should not impact the others. This can be achieved by isolating services from each other and implementing proper error handling and timeouts.
6. **Infrastructure Automation**: Use of automation tools for testing, deployment, and scaling of microservices. This is commonly facilitated by CI/CD practices and container orchestration platforms like Kubernetes.
7. **Design for Replaceability**: Services should be designed to be replaceable. If a service is failing or needs to be updated, it should be possible to replace it without disrupting the entire system.
8. **Observability**: It should be possible to monitor the system as a whole and each individual service. This includes logging, monitoring, and tracing capabilities to understand the behavior of services and diagnose issues.
9. **Loose Coupling**: Services should have as few dependencies on each other as possible. This means changes to one service should require minimal or no changes to another.
10. **High Cohesion**: The related functionalities should be kept within the same service. This means the code related to a specific functionality should not be scattered across multiple services.

These principles are meant to guide the design and implementation of microservices, enabling a system that is flexible, scalable, and easier to manage and evolve over time.

Top of Form

Bottom of Form

## Difference between SOA and Microservices

One key difference between Microservices architecture and Service-Oriented Architecture (SOA) is the depth of their decomposition, which refers to how components of a system are broken down into smaller parts.

**SOA** typically decomposes at the service level. That means it divides a system into individual services, each of which can be used and reused to support different applications. These services communicate with each other, often through a service bus that handles data exchange. While services in SOA are often large and multifunctional, they are still intended to be reusable and can be exposed to different applications.

On the other hand, **Microservices** architecture decomposes even further, down to the level of an individual service's execution environment. This means that each microservice is not only a separate service, but it also has its own runtime environment and can be developed, deployed, and scaled independently of the others. Each microservice is highly cohesive, performing a single business function.

This deeper level of decomposition in microservices architecture can lead to greater flexibility, as each service can use its own technology stack and can be updated without affecting other services. However, it also adds complexity in terms of managing, deploying, and monitoring the individual services.

## A key distinction between monolithic applications and microservices, particularly when dealing with Java applications.

In a **monolithic** application, you typically package your entire application into a single WAR (Web Application Archive) or EAR (Enterprise Application Archive) file. This bundle is then deployed onto a Java Enterprise Edition (JEE) application server like JBoss, WebLogic, or WebSphere. Multiple applications can be hosted on the same server, and they all share the resources provided by that server.

On the other hand, with the **microservices** approach, each service is built as a 'Fat JAR' (Java Archive), often using a framework like Spring Boot. This JAR file includes not only the service itself but also all of its dependencies, which can include an embedded server. This setup means each microservice can run as its own standalone Java process, independent of other microservices.

This fundamental difference embodies the principles of microservices - individual components are isolated, independently deployable, and can run in their own environments. This provides advantages like improved fault isolation, scalability per service, and technology diversity, but also introduces complexity in terms of service orchestration and inter-service communication.

## Characteristics of microservices

### **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/MQTT), HTTP, and REST are commonly used for interaction means communication 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.

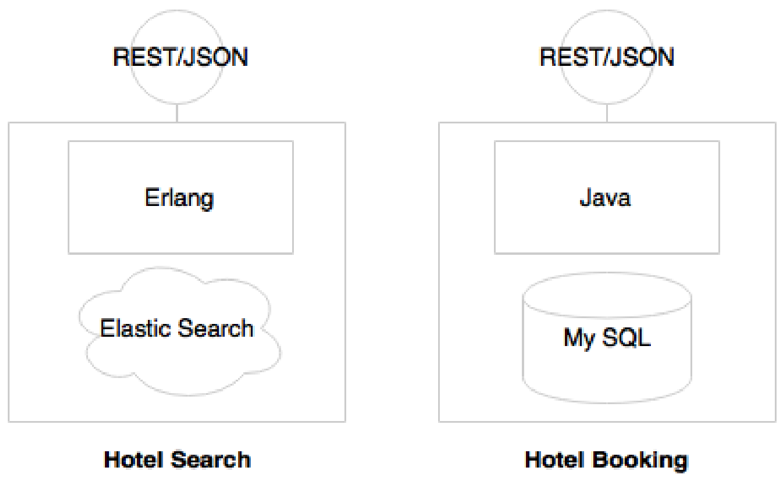
Microservices are discoverable.

### **Microservices are lightweight**

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

Preferred to use **Docker containers** instead of VMs to help keep the infrastructure footprint as minimal as possible.

### **Microservices with polyglot architecture**

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

### **Automation in a microservices environment**

As microservices break monolithic application into many 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 automated Infrastructure



The development phase is automated by using version control tools such as Git together with continuous Integration (CI) tools such as Jenkins.

The testing phase will be automated by using testing tools such as selenium.

Automated deployments are handled by using DevOps.

Infrastructure provisioning is done through Cloud.

### **Microservices ecosystem**

Microservices implementations have a supporting ecosystem including DevOps, Centralized log management, Service registry, API gateways, Service routing, Flow control mechanism



### **Microservices are distributed and dynamic**

**Distributed**: Microservices are designed to run on different machines or containers, which can be located in various parts of a network or even distributed across multiple data centers or cloud providers. This distributed nature is beneficial for resilience and scalability, as individual services can fail or be scaled without impacting others. However, it also introduces complexity and challenges in terms of network communication, data consistency, and system coordination.

**Dynamic**: Microservices are dynamic in nature, in that they can be independently deployed, updated, scaled, and retired without affecting the whole system. Additionally, with the aid of container orchestration tools like Kubernetes or Docker Swarm, new instances of a service can be spun up or taken down in response to demand, and services can be moved around within the system as resources change. This dynamic nature improves the system's flexibility and responsiveness, but also requires robust service discovery and routing mechanisms to ensure requests reach the correct service instances.

The distributed and dynamic nature of microservices are key to the benefits they provide, such as resilience, scalability, and flexibility. But they also introduce challenges that need to be managed, such as service coordination, data consistency, network latency, and system observability.

## Microservices benefits

Microservices architecture provides several significant benefits:

1. **Scalability**: Each microservice can be scaled independently according to its needs, which can lead to more efficient use of resources. If a particular service has high demand, only that service can be scaled up without affecting others.
2. **Speed of Development and Deployment**: As each microservice is independent, it can be developed, tested, and deployed independently. This enables faster and more frequent updates and reduces the risk of the entire application being affected by a single update.
3. **Fault Isolation**: If a failure or a bug is introduced in a microservice, the problem is isolated to that service and does not directly affect others. This leads to higher overall system availability.
4. **Technological Freedom**: Different microservices can be developed using different programming languages, databases, and software environments. This allows teams to choose the technology stack that is best suited for the service's requirements.
5. **Productivity**: Microservices can be developed by small teams that take ownership of their service. This can lead to increased productivity as the team can focus on a specific task without worrying about the whole application.
6. **Ease of Understanding**: Since each microservice focuses on a specific business capability, they are smaller and simpler to understand, making it easier for new developers to understand and contribute to the service.
7. **Reusability**: Microservices are modular and can be reused across different projects or applications.

## Microservices early adopters

While microservices provide these benefits, it's also important to note that they introduce complexity in terms of service coordination, data consistency, network communication, and observability. Therefore, the decision to use a microservices architecture should be based on a thorough understanding of the system requirements, the organizational capabilities, and the trade-offs involved.

Several prominent technology companies adopted microservices architecture early on and have shared their experiences publicly, thereby promoting the approach and its benefits. Some of these early adopters include:

1. **Netflix**: Netflix is perhaps one of the most famous early adopters of microservices. The global streaming service transitioned from a monolithic architecture to microservices to support its rapidly growing user base and ensure high availability and resilience.
2. **Amazon**: Amazon moved to a microservices architecture in the early 2000s as part of a significant architectural shift. This enabled Amazon to scale their operations massively and develop services like AWS.
3. **Ebay**: Ebay moved from a monolithic architecture to a more distributed, service-oriented architecture in the mid-2000s to support its growing user base and rapidly changing business requirements.
4. **Twitter**: Originally built as a monolithic Ruby on Rails application, Twitter transitioned to a microservices architecture to handle its massive scale and ensure high availability.
5. **Uber**: As Uber’s user base and geographic reach grew, it moved to a microservices architecture to support its need for speed, scalability, and innovation.
6. **Spotify**: Spotify adopted a model of "squads", "tribes", and "guilds" to build its services independently, aligning with the principles of microservices.

These early adopters have shown the potential of microservices to support large-scale, rapidly changing, and highly available systems. Their experiences have also highlighted the challenges and complexities involved in adopting microservices and have led to the development of various tools, practices, and patterns to manage these challenges.

Top of Form

Bottom of Form

**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 of jetty). Spring boot is a tool to develop such kinds of services i.e., Spring Boot enables microservices development by packaging all the required runtime dependencies in an executable fat Jar file.

1. **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 unit remains manageable.
3. Polyglot Architecture: If different requirements need different architectures, different technologies, etc. then split them as separate Microservice.
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 flight 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: Microservice 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 can 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, the http listener such as tomcat or jetty of jboss, etc is needed but not messaging listener. 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 request has been processed.
2. Adds **hard dependencies** (tight coupling) 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 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 listener.
2. It is complex to handle the fault tolerance of a messaging listener.

**How to decide which style to choose?**

It is not possible to develop a system with just approach. A combination of both approaches are required based on the use cases. In principle, the asynchronous approach is great for microservices. 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, wait and read response from the output queue. 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 fulfill 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 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 schemea, 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.

**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.



1. **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 development 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 Elastic search. 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.

**Organization culture**

One of the biggest challenges in microservices implementation is the organization culture.

Organization 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.

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.

1. **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 micro services 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 micro services, audit services, replication services, or external applications, and the like. This allows other microservices and applications to respond to state changes.
* **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, use 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 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 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 victual machines. VMW are, 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 life cycle 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 the 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 spigot could add value to microservices monitoring.
* **Dependency and management**: We also need tools to discover runtime topologies, service dependencies, and to manage configurable items. A graph-based CMOB 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 of 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 micro services 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.

1. **MICROSERVICES EVOLUTION- A CASE STUDY**

We will discuss Brown field Airline and their journey from a monolithic passenger sales and service(pss) application to 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 able to implement for microservices such as fair, 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 Rabbit MQ. In this implementation, the **Oracle database will be used with separate schema for each microservice**. The code is section demonstrates all the capabilities highlighted in green colour above.



The following steps are used to set up PSS micro services project

* 1. Create Tablespace, schemas, tables, sequences and insert data by referring ‘documents/misc/Airlines\_PSS\_schema.doc’ file.
  2. Download STS from https://spring.io/tools/sts/all. Start STS (spring tool suite) and select ‘**Micro services work space’** from the back up.
  3. Start **FaresFlightticket** by right click and run as **Spring boot app.**
  4. Install RabbitMQ server from software’s 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. Hence install OTP\_win64\_19.3.exe from software folder.
  5. Start SearchFlighTickets by right click and run as **Spring boot app**.
  6. Start BookingFlightTickets by right click and run as **Spring boot app.**
  7. Start CheckingCustomers by right click and run as **Spring boot app.**
  8. Start FlightWebsite by right click and run as **Spring boot app.**

Each service has multiple packages and there 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.

|  |  |  |  |
| --- | --- | --- | --- |
| **Microservice Name** | **REST endpoints synchronous** | **Messaging endpoints asynchronous** | **Used by** |
| FareFlightTicket | http://localhost:8081/fare/{} |  | Booking Microservices |
| SearchFlightTicket | <http://localhost:8090/search/get> |  | Website |
| SearchFlightTickets |  | @RabbitListener(queues=”**inventoryQ**”) | Search Microservice itself subscribed to  **inventoryQ** for inventory updates. |
| BookingFlightTickets | <http://localhost:8060/booking/create> |  | Website |
| BookingFlightTickets | [http://localhost:8060/booking/get/{id}](http://localhost:8060/booking/get/%7bid%7d) |  | Checkin,website |
| BookingFlightTickets |  | Template.convertAndSend(“**inventoryQ**”,message); | Search microservice |
| BookingFlightTickets |  | @RabbitListener(Queues=”checkInQ”) | BookingService subscribed to **CheckinQ** for check-In updates |
| CheckInCustomers | <http://localhost:8070/checkin/create> |  | Website |
| CheckInCustomers | [http://localhost:8070/checkin/get/{id}](http://localhost:8070/checkin/get/%7bid%7d) |  | Not used |
| CheckInCustomers |  | Template.convertAndSend(“**checkinQ**”,message); | Booking Microservice |

We have accomplished the following item in implementation so far.

1. Each microservice exposes a set of REST/JSON endpoints for accessing business capabilities.
2. Each micro service implements certain business functions using the spring framework.
3. Each micro service has its own schema in Oracle database.
4. Micro services 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. And OAuth2-based security mechanism is developed to protect the microservices.